AFRL-RI-RS-TR-2008-205
Final Technical Report
July 2008

DEVELOPMENT OF ENHANCED INTERACTIVE DATAWALLS FOR DATA FUSION AND COLLABORATION

University of Alabama

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2008-205 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/

ALEX SARNACKI
Work Unit Manager

/s/

JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2008	Final	Mar 04 – May 08

4. TITLE AND SUBTITLE

DEVELOPMENT OF ENHANCED INTERACTIVE DATAWALLS FOR DATA FUSION AND COLLABORATION

5a. CONTRACT NUMBER
FA8750-04-C-0067

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69329W

6. AUTHOR(S)

Jingyuan Zhang

5d. PROJECT NUMBER
NASA

5e. TASK NUMBER
BA

5f. WORK UNIT NUMBER
02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Alabama
Office for Sponsored Programs
801 University Blvd.
Tuscaloosa, AL 35487-0104

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RISB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RISB

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-205

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-4183

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Develop interactive display technology for multiple users to collaborate and manage information through a high resolution extended display Datawall using the applications they are familiar with.

15. SUBJECT TERMS
Datawall, Collaboration, Multi-user, Interactive, Display, Multi-cursor, Multiple User, Multiple Cursor

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alex Sarnacki
U	U	U	UU	47	19b. TELEPHONE NUMBER (*Include area code*) 315-330-4985

Table of Contents

List of Figures

1. Introduction

A command and control room can consist of wall-sized computers, referred to as DataWalls hereafter, for group use as well as desktops/laptops for individual use. Information sharing among these computers and easy interaction with these computers are two crucial issues in such an environment. In this project, a suite of software has been developed to address these two issues. Specifically

- A user can move an application from one computer to another for viewing and interaction. Both VNC(Virtual Network Computing)-based and mirror-driver-based approaches are used to implement this functionality.
- A user can interact with a DataWall using camera-tracked laser pointers.
- A user can move a cursor from one computer to another for multiple computer interaction and annotation.
- A user can perform copy and paste operations between two computers.

In this report, we will detail how these functions are implemented.

2. VNC-based Approach

VNC (Virtual Network Computing) was developed by a group of researchers at AT&T Laboratories in Cambridge, UK. It is open-source software. VNC allows a user to view and interact with a remote computer (known as the VNC server) from a local computer (known as the VNC viewer/client). We use VNC to display multiple remote desktops on the DataWall. We start with the original VNC for desktop sharing. We then modified it for application sharing.

2.1 VNC Basics

The VNC 4.0 server can be started in user mode or service mode. Regardless of the starting node, it ends up starting a new process that runs *winvnc4.exe* compiled from *winvnc.cxx*. In the *main* function of *winvnc.cxx*, a variable *server* of type *VNCServerWin32* is defined, and its *run* function is executed. In the *VNCServerWin32*'s *run* function, an instance *rfb* of class *ManagedListener* is created and its *setPort* function is executed. In the *ManagedListener*'s *setPort* function, a new instance of *TcpListener* is created based on the port number. (In the *TcpListener*'s constructor, the necessary *socket*, *bind*, and *listen* functions are called. The *WSAStartup* function is called by *network::TcpSocket::initTcpSockets* in the *main* function.) The newly created *TcpListener* instance, thereafter referred to as the *listener socket*, is added to *sockMgr* of type *SocketManager* in *VNCServerWin32* using the *addListener* function. That is the first socket added to *sockMgr*. (Later, whenever a client is connected to the server, a new socket, referred to as a *client socket*, will be added to *sockMgr* by using the *addSocket* function.)

All the sockets in *sockMgr* are monitored by the *getMessage* function of *SocketManager* through their corresponding events. In fact, the *getMessage* function does all the monitoring jobs. In addition to monitoring all the sockets, *getMessage* also monitors whether the server desktop has changed via its *updateEvent* (added to *sockMgr* using the *addEvent* function in the *VNCServerWin32* constructor) and checks all the window messages in the queue as specified by *QS_ALLINPUT* in *MsgWaitForMultipleObjects*. If the window message is WM_QUIT, it returns false. For the other types of window messages, it returns the message as one of its parameters.

If there is an *FD_READ* (or *FD_CLOSE*) network event on a client socket, the *getMessage* function of *SocketManager* calls the *VNCServerST*'s *processSocketEvent* function which first finds the corresponding client of type *VNCSConnectionST* in its *clients* list (This is needed because the software does not take advantage of the one-to-one relationship between a client and a socket.) and then calls the *VNCSConnectionST*'s *processMessages* function. The *processMessages* calls the *processMsg* which will process the incoming message based on the current state and the RFB protocol. The specific code is listed below. The first four cases are for the initial handshaking, and the last case is for the normal communication after authentication.

```
switch (state_) {
case RFBSTATE_PROTOCOL_VERSION: processVersionMsg();      break;
case RFBSTATE_SECURITY_TYPE:    processSecurityTypeMsg(); break;
case RFBSTATE_SECURITY:         processSecurityMsg();     break;
case RFBSTATE_INITIALISATION:   processInitMsg();         break;
case RFBSTATE_NORMAL:           reader_->readMsg();       break;
}
```

If an *FD_ACCEPT* (or *FD_CLOSE*) network event is identified on a listener socket in *getMessage*, *getMessage* calls the *accept* function to accept the connection from a new client, and adds the newly created client socket to *sockMgr* by using the *addSocket* function. At the same time, a new client of type *VNCSConnectionST*, corresponding to the newly created client socket, is added to the *clients* list of *VNCServerST* by the *addClient* function. The *addClient* function also calls *VNCSConnectionST*'s *init* function which calls the *initialiseProtocol* function to send the server version to the client and to set its state to `RFBSTATE_PROTOCOL_VERSION`. The initial handshaking will follow as specified in the above *switch* statement.

When a client is authenticated successfully, the *authSuccess* function of *VNCSConnectionST* will be called. If it is the first client, the *desktop* of type *SDisplay* in *VNCServerWin32* will be started by calling the *start* function. The *start* function calls *WMHook*s' *setUpdateTracker* function. The *setUpdateTracker* function calls the global *addHook* function which, in turn, calls the global *StartHookThread* function. In the *StartHookThread* function, a thread called the *hook owner thread* of type *WMHooksThread* is created and the global *WM_Hooks_Install* function is called to install three hooks: *HookCallWndProc* for *WH_CALLWNDPROC* (responding to the messages

sent to a window), *HookGetMessage* for *WH_GETMESSAGE* (responding to the messages posted to a message queue), and *HookDialogMessage* for *WH_SYSMSGFILTER* (responding to the messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar).

When any of the three hooks described in the previous paragraph is called, it calls the *ProcessWindowMessage* function. Based on the received message type, it calls the *NotifyWindow*, *NotifyWindowClientArea*, *NotifyWindowBorder*, *NotifyRectangle*, or *NotifyCursor* function which, in turn, calls the *NotifyHookOwner* function. *NotifyHookOwner* notifies the hook owner thread using *PostThreadMessage* with one of the five user-defined messages: *WM_HK_WindowChanged*, *WM_HK_WindowClientAreaChanged*, *WM_HK_WindowBorderChanged*, *WM_HK_RectangleChanged*, and *WM_HK_CursorChanged*. When the hook owner thread receives the message, it calls the global *NotifyHooksRegion* function which passes the change to *change_tracker* of type *SimpleUpdateTracker* in *SDisplay* using the user-defined message type WM_USER. After the change is made, the *updateEvent* in *SDisplay* is set by the *triggerUpdate* function. The *getMessage* function of *SocketManager*, described in an earlier paragraph, will detect the event, and invoke *SDisplay*'s *flushChangeTracker* to pass the change to *comparer* of type *ComparingUpdateTracker* in *VNCServerST*.

When the server receives the *msgTypeFramebufferUpdateRequest* message from a client, the *framebufferUpdateRequest* function of the corresponding *VNCSConnectionST* will be called. Once the *requested* of type *Region* in *VNCSConnectionST* is updated, the *writeFramebufferUpdate* of *VNCSConnectionST* is called. The *writeFramebufferUpdate* function first calls *VNCServerST*'s *checkUpdate* to pass the change saved in *comparer* of *VNCServerST* to every client (i.e. to *updates* of type *SimpleUpdateTracker* in *VNCSConnectionST*). Based on the *requested* and the *updates*, the actual update is computed and sent from the server to the client. The *FramebufferUpdate* message is from server to client. It consists of a sequence of rectangles of pixel data for the client to update its frame buffer. Each rectangle of pixel data may be encoded. There are six encoding schemes available: *Raw*, *CopyRect*, *RRE*, *CoRRE*, *Hextile*, and *ZRLE*.

The VNC transfers rectangles of pixel data from the server to the client/viewer. Four different pixel buffers are involved in the pixel transfer. Figure 2.1 illustrates the pixel flow among those four buffers. When the server receives the *msgTypeFramebufferUpdateRequest* request from the client, the server retrieves the pixels of the specified rectangles from the frame buffer to its internal buffer, and sends these rectangles of pixels to the client as the reply using the *msgTypeFramebufferUpdate* message. When the client receives a rectangle of pixels, it saves the pixels to its internal buffer, and calls the *InvalidateRect* function which causes a WM_PAINT window message to be sent. When handling the WM_PAINT message, the client copies the pixels from its internal buffer to the frame buffer. This is how the changes/updates on the server's desktop are reflected in the VNC viewer's window on the client computer. The

client makes the first *msgTypeFramebufferUpdateRequest* request when it is authenticated, and makes each subsequent request after it finishes handling a WM_PAINT message.

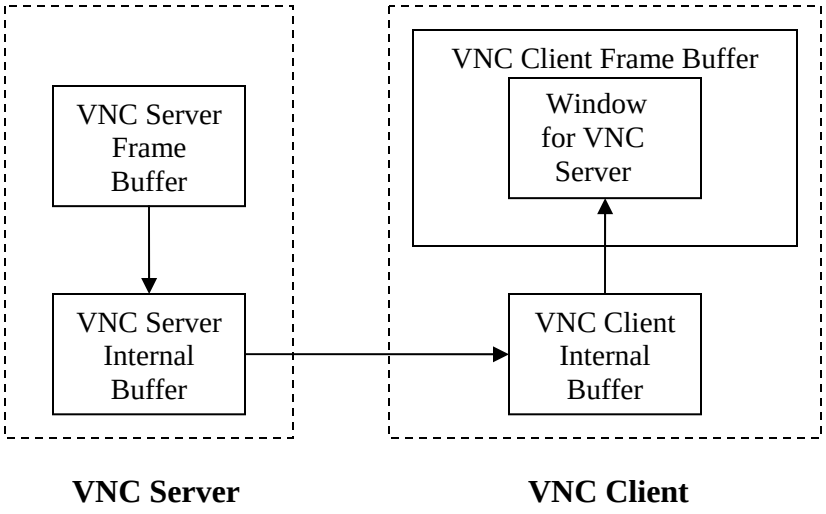

Figure 2.1 The pixel flow among 4 pixel buffers in the VNC.

2.2 Modifications to VNC 4.0 for Application Sharing

The original VNC software can only project the whole desktop from a server to the DataWall. We modified the VNC server and viewer to project the whole desktop or an application of interest from a server to the DataWall. In the modified version, we used a global variable called *TargetProcessID* to distinguish between projecting a single application and projecting the whole desktop. If *TargetProcessID* is zero, the whole desktop will be projected. Otherwise *TargetProcessID* is the process identification of the application that will be projected. Projecting a single application is much harder than projecting the whole desktop because a single application occupies only a part or even none of the desktop. In the desktop projecting, the changes can be always additive (or aggregated) whereas the changes in the application projecting can be additive or subtractive. For example, in the desktop projecting, when a window is moved, the changes include the window at the new position plus all the newly exposed areas that were behind the window before the movement. Yet in the application projecting, the changes include adding the window at the new position and subtracting the newly exposed areas. To address this issue, we adopt the absolute approach for the application projecting and keep the relative approach for the desktop projecting. In the absolute approach, no hooks are used to collect the changes. (Hooks are considered as inefficient because they are installed in all the applications running on the desktop.) To indicate

whether the rectangles of pixels are relative or absolute, we added a new message type called *msgTypeMode* (100) to be sent from the server to the client after authentication.

We also modified the protocol to allow an *msgTypeFramebufferUpdate* message containing no rectangle to be sent. Hooks in the desktop sharing are so inefficient that there are always new changes between two consecutive *msgTypeFramebufferUpdateRequest* requests, which keeps the communication between the client and the server going. In the application projecting, when an application is minimized, no window will be visible. Therefore no *msgTypeFramebufferUpdate* message will be sent to the viewer based on the original protocol, and no new *msgTypeFramebufferUpdateRequest* request will be automatically issued from the viewer. The communication between the viewer and the server stops. When the application at the server is restored, the viewer will not know it. The communication can only be restored by sending the *msgTypeFramebufferUpdateRequest* message at the request of the user manually. In our modified VNC software, we allow the *msgTypeFramebufferUpdate* message with zero rectangles. After the viewer receives such a message, a new request is issued immediately to keep the communication active.

In the application projecting, an application running on a remote computer can be projected on the DataWall with its own window frame. The VNC viewer does not add its frame to the application. Therefore, there is no difference between a projected application from a remote computer and an application started on the local DataWall computer. To achieve this, we added a new message called *msgTypeVisibleWindows* (99) to the VNC protocol in the server to viewer direction. For the application projecting, in addition to using the *msgTypeFramebufferUpdate* message to send rectangles of pixels, we use the new message type *msgTypeVisibleWindows* to send the information about the windows involved. The information for a window is described in the class named *CSVisibleWindow*:

```
class CSVisibleWindow {
   unsigned int m_nWid  // window ID
   unsigned int m_nPid  // parent window ID
   Rect m_rRect;        // window geometry
};
```

Therefore, the *msgTypeVisibleWindows* message completely describes the relationship among those related windows. In addition, the server orders the windows in the *msgTypeVisibleWindows* message based on their stack order. Based on the *msgTypeVisibleWindows* message received from the server, the viewer can reconstruct the windows from the message.

To eliminate the flicking effect, we use double window chains at the viewer side by defining the following three variables in the *CView* class:

```
bool bIsZeroActive;     // Is visibleWindows0 active?
std::vector<CCVisibleWindow> visibleWindows0;
std::vector<CCVisibleWindow> visibleWindows1;
```

Whenever the viewer receives the information about a window, it will add the window to the active visible window vector. Before adding the window, it first checks whether the window already exists at the viewer side by searching in the inactive (previous) vector. If the window exists in the previous vector, the window will be copied to the active vector. Otherwise, the client creates a window of type *CVNCWnd*. If the window to be created has a non-zero parent window ID, the corresponding parent window at the viewer side can be found from the active vector. Since the windows are ordered by their stack order, the algorithm will not fail. All the windows in the previous vector that are not copied (i.e. they no longer exist) will be destroyed. Finally, every window in the active vector will be painted by calling *MoveWindow* if the window has been resized or *InvalidateRect* otherwise. The actual painting is done by copying the pixel data stored in *buffer* of the VNC main window of type *CView*. In the application projecting, the main VNC window still exists, but it is not visible.

2.3 Visitor and Local Modes for Application Sharing

The original VNC is only capable of desktop sharing. The modified VNC can perform application sharing in addition to desktop sharing. For application sharing, how to select an application for sharing is a key issue. To address both privacy and convenience issues in application sharing, a VNC server can be run in one of the two modes*: local* and *visitor*.

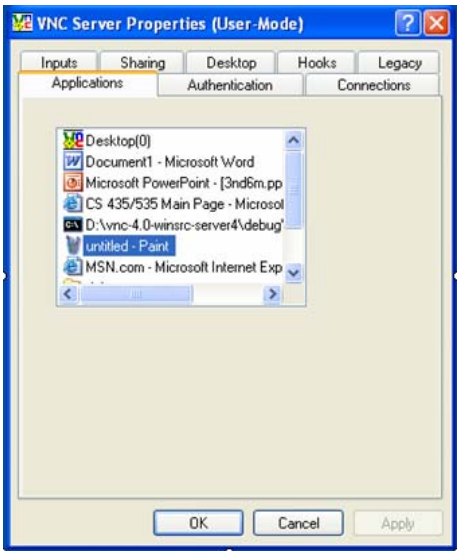

Figure 2.2 Selection of an application at the application computer for sharing.

In the visitor mode, the user (i.e. the visitor) at the application computer end is responsible for selection of an application for sharing. The visitor can select any application running on his/her computer for sharing on the data wall using the property sheet as shown in Figure 2.2. A user at the Data Wall is not able to select an application

of the application computer to be shared. Once an application is selected for sharing, the application window(s) will be projected on the DataWall, and a user at the DataWall end can take control of the application. However, once an application window is minimized, the data wall user is not able to restore it. When an application is minimized within Windows, an icon is shown on the task bar. A minimized application can be restored by clicking the corresponding icon. However, the task bar is not projected in application sharing. To address the issue, we added a task box that lists all minimized windows of the shared application in the visitor mode. Figure 2.3 shows a task box in the visitor mode.

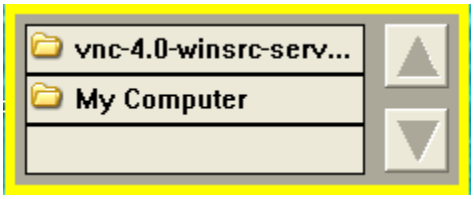

Figure 2.3 A task box in the visitor mode.

In the local mode, the user at the DataWall end takes full control of the remote computer. The user at the DataWall end can launch any application and select any application for sharing without physically accessing the remote computer. The magic is again the task box. Figure 2.4 shows a task box in the local mode. It consists of two parts: the launch bar and the task list. The DataWall user uses the launch bar to launch an application at the remote computer. There are four applications shown in the figure, Windows Explorer, MS Paint, Internet Explorer, and MS Word. To start MS Paint, Internet Explorer or MS Word, the user just clicks the corresponding icon. If the user wants to start other applications, the user needs to first launch the Windows Explorer and select the Windows Explorer for sharing. From the shared Windows Explorer, the user can go over the whole file system and start any application from the Windows Explorer. In the local mode, the task list consists of all the applications running on the remote computer. In fact, the applications shown in the task list here is equivalent to the applications shown in the property sheet of Figure 2.2. Since the space is very limited on the DataWall, only three application windows are shown directly at one time, the other application windows can be accessed using the up and down arrow keys.

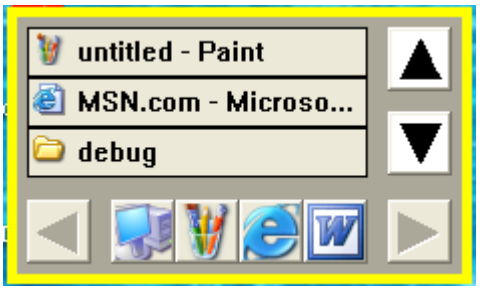

Figure 2.4 A task box in the local mode.

In both local and visitor modes, we need to send a list of window icons and titles from the VNC server to the VNC client (i.e. the DataWall) and display them on the task list in the task box. In the local mode, the list consists of all application windows running on the remote computer. In the visitor mode, the list consists of the minimized windows that belong to a shared application. To transfer the list of window icons and titles, we added a new message type named *msgTypeAppIcons* to the VNC protocol. The message starts with the number of windows. For each window, it consists of the window id, window title and window icon. The most difficult part is how to transfer the icon from the VNC server to the VNC client. An icon consists of two small images. These two images are device-dependent bitmaps (DDB). Therefore we have to covert device-dependent bitmaps to device-independent bitmaps (DIB) at the server end before sending them to the client. At the client end, we need to convert DIBs to DDBs compliant to the client format, and make an icon from two DDBs. Finally, the icon will be shown with the title in the task list. The icons in the launch bar are also sent from the server using the *msgTypeAppIcons* message type.

To implement the transfer of window information including window ID, tile and icon from the server to the client, we defined a class called *CSTaskWindow* at the server and *CCTaskWindow* at the client. The data members of *CCTaskWindow* are listed below, with *CSTaskWindow* being similar.

```
class CCTaskWindow {
   unsigned int Id;  // window handle
   char *m_pTitle;   // window title
   int IconWidth;          // icon width
   int IconHeight;         // icon height
   int IconWidthBytes;     // icon width in bytes for bitmask bitmap
   unsigned char *m_pIconMask;   // DIB data for icon bitmask bitmap
   unsigned char *m_pIconColor;  // DIB data for icon color bitmap
   HICON m_hIcon;                // icon handle
};
```

At the server side, we defined *FindIconInfo()* for *CSTaskWindow* to get icon information including *IconWidth*, *IconHeight*, *IconWidthBytes*, *m_pIconMask* and *m_pIconColor* from the *m_hIcon* handle. These five pieces of information will be sent from the server to the client. At the client end, we defined *MakeIcon()* for *CCTaskWindow* to use the received icon information to make an icon with the *m_hIcon* handle. Both *m_pIconMask* and *m_pIconColor* are DIB image data. In our implementation, the bitmask bitmap has a depth of one-bit, and the color bitmap has a depth of 32 bits. When an image has a depth of one-bit, there is no difference between its DDB format and its DIB format. Since the image data for a scan line must be word-aligned, sometimes padding is necessary. *IconWidthBytes* indicates the number of bytes in a scan line. The *GetBitmapBits()* function is used to get *m_pIconMask* from an *HBITMAP* handle which is in turn obtained from an *HICON* handle by using the *GetIconInfo()* function. The DDB and DIB conversion for a color bitmap is performed by

using the *GetDIBits()* and *SetDIBits()* functions. Finally an icon is made by calling the *CreateIconIndirect()* function.

We also added the *msgTypeCommand* message to send a command from the client to server. The message consists of three command subtypes related to the task box: restoring a minimized window in the visitor mode, launching an application and selecting an application for sharing in the local mode.

2.4 Reverse Connection in VNC

A connection is normally established from the viewer (i.e. the DataWall) to a server (i.e. a remote application computer). The VNC also allows the reverse connection, i.e. the connection initiated from a remote application computer to the DataWall. The reverse connection allows a user to send an application from an individual application computer to the DataWall.

When *vncviewer.exe* is started with the *–listen* option, it becomes a daemon, i.e. a server. In the *WinMain* of the VNC viewer, the *WSAStartup* function is called by *TcpSocket::initTcpSockets()* regardless of the connection direction. If the *–listen* option is used with the VNC viewer, the *acceptIncoming* variable will be set to true, and the necessary *socket*, *bind*, and *listen* functions will be called in the constructor of *network::TcpListener* which is constructed in the *addDefaultTCPListener()* function of *view_manager* of type *CViewManager*. The *addDefaultTCPListener()* function has a parameter named *port* on which the daemon will be listening. A connection request from a remote application computer (*FD_ACCEPT*) will be converted to a *WM_USER* window message to *CViewManager* by the *WSAAsyncSelect* function. When the *processMessage()* function of *CViewManager* processes the *WM_USER* message because of a connection request from a remote application computer, it calls the *accept* function and then the *start()* function of *CViewThread*. The *start()* function of *CViewThread* creates a thread to execute the *threadProc()* function which in turn calls the *run()* function. In the *run()* function of *CViewThread*, a *view* of type *CView* is defined, and the *initialise()* function of *CView* is called. The *initialise()* function calls the *initialiseProtocol()* function to initialize the connection state to *RFBSTATE_PROTOCOL_VERSION* and uses *WSAAsyncSelect* to convert the read (*FD_READ*) or close (*FD_CLOSE*) event of the connection to the *WM_USER* message to be processed by *CView*. The initial handshaking between the daemon and the remote application computer is handled by processing the *WM_USER* messages.

What happens at the remote computer side is a little complicated. This is mainly because *winvnc.exe* can be started in user mode and service mode. Here we are only interested in the user mode. However, the user mode code was written to mimic the service mode. There are two steps to establish a connection to the VNC viewer daemon from a remote application computer. The first step is to execute *winvnc.exe* to start the regular VNC server. The second step is to execute *winvnc.exe* again with the *–connect*

option to make a connection request by specifying the host name and port number of the VNC viewer daemon. The connection is not established directly. The second instance of *winvnc.exe* makes a request to the first instance of *winvnc.exe*, and the first instance (i.e. the regular VNC server) makes a connection request to the VNC viewer daemon on behalf of the first instance. Specifically, when the second instance starts with the *–connect* option, it uses the *FindWindow* function to find the tray icon of the first instance and send the tray icon a message of type *WM_COPYDATA*. The message carries the host name and the port number of the VNC viewer daemon. When the tray icon of the first instance receives the *WM_COPYDATA* message, it calls the *addNewClient()* function of the VNC server (i.e. *VNCServerWin32*) with the host name and the port number. The *addNewClient()* function constructs a *TcpSocket* to connect the VNC viewer daemon with the host and port provided, and calls the *queueCommand()* function to queue the *AddClient* command with the constructed socket. When the *AddClient* command is processed by the *doCommand()* function of *VNCServerWin32*. The *addClient()* function of *vncServer*, declared as a *VNCServerST* , will be called to add a reverse connection to the VNC server. The rest will be the same as the connection from the VNC viewer to the VNC server.

2.5 VNC for Unix

The VNC code for UNIX is similar to the code for Microsoft Windows. In fact, the codes for both operating systems share a majority of components including *network*, *rdr*, *rfb*, *Xregion* and *zlib*. In VNC 4.1.2 for UNIX, there are two directories, *common* and *unix*. We installed the VNC server as an extension to XFree86 4.3.0 under Fedora 8. Figure 2.5 illustrates the communications among three main programs: VNC Extension, *vncviewer* and *vncconfig*. The report will describe what is unique to our installation. Briefly, the VNC server is loaded as a module (*vnc.so*) that extends the X server. As an X client, *vncconfig* configures and controls the VNC server using the VNC extension to the X protocol. *vncviewer* communicates with the VNC server using TCP/IP sockets.

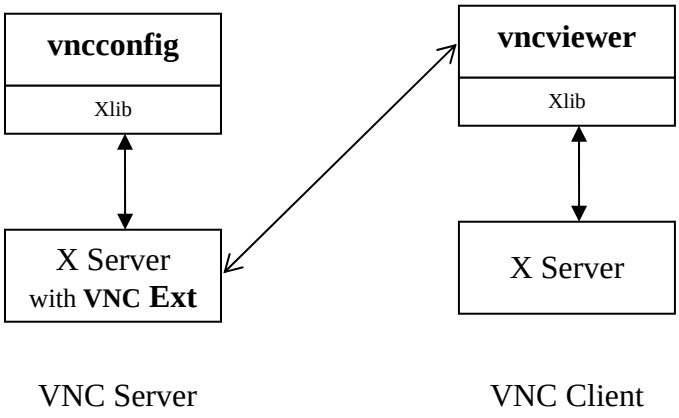

Figure 2.5 The communications among three main VNC programs.

Module Loading and Initialization

To load *vnc.so*, a line of `Load "vnc"` was added to the *"Module"* section of the */etc/X11/xorg.conf* configuration file. In the *xf86vncModule.cc* file, the following line indicates the module version information being *vncVersRec* and the module setup function being *vncSetup*.

```
XF86ModuleData vncModuleData = { &vncVersRec, vncSetup, NULL };
```

The *vncSetup* function calls *LoadExtension* with *vncExt* of type *ExtensionModule* as the parameter. *vncExt* specifies *vncExtensionInitWithParams* as the extension initialization function. *vncExtensionInitWithParams* first retrieves the VNC-related options in the *"Screen"* section of the */etc/X11/xorg.conf* configuration file and sets the retrieved option values as the values for the corresponding parameters. One of the retrieved option values is the name of the file that stores the VNC server password. Then *vncExtensionInit* in *vncExtInit.cc* is called. *vncExtensionInit* calls *AddExtension* in the following format to add the VNC extension named *VNCEXTNAME*. Later *vncconfig* can make requests to the VNC extension to be processed by *ProcVncExtDispatch* or *SProcVncExtDispatch*, depending on whether the byte order needs to be swapped.

```
AddExtension(VNCEXTNAME, VncExtNumberEvents, VncExtNumberErrors,
             ProcVncExtDispatch, SProcVncExtDispatch, vncResetProc,
             StandardMinorOpcode);
```

Then *vncExtensionInit* will set up a listener to wait for connections from VNC viewers and call *vncHooksInit* to hook up drawing-related functions.

Drawing Hookup

A regular X client (or application) uses the X protocol to ask the X server to perform the drawing on the screen. The *vncHooksInit* function in *vncHooks.cc* is responsible for intercepting the drawing-related functions and figuring out the affected region on the screen. The affected region will then be sent to the VNC viewer.

The drawing-related functions are divided into three groups. The first group consists of the screen-related functions including `CloseScreen, CreateGC, PaintWindowBackground, PaintWindowBorder, CopyWindow, ClearToBackground, RestoreAreas, InstallColormap, StoreColors, DisplayCursor,` and `BlockHandler`. The second group consists of the GC(Graphics Context)-related functions including `ValidateGC, ChangeGC, CopyGC, DestroyGC, ChangeClip, DestroyClip,` and `CopyClip`. The third group consists of the GC-related operations including `FillSpans, SetSpans, PutImage, CopyArea, CopyPlane, PolyPoint, Polylines, PolySegment, PolyRectangle, PolyArc, FillPolygon, PolyFillRect, PolyFillArc, PolyText8, PolyText16, ImageText8, ImageText16, ImageGlyphBlt, PolyGlyphBlt,` and `PushPixels`. The *vncHooksInit* function first saves the original drawing-related functions, and replaces them with the

corresponding VNC functions that have a *vnchook* prefix. Therefore, whenever, an X client causes a drawing-related function to be called, the corresponding VNC function will be called. In the corresponding VNC function, the saved original drawing function will be first called to perform the actual drawing. Then any screen changes caused by the drawing function will be recorded into the VNC server, and sent to the VNC viewers.

The vncconfig Program

The *vncconfig* program is an X client that communicates with the VNC extension to the X server to configure and control the VNC server. It first calls the *XInitExtension* function with *VNCEXTNAME* to find the `major_opcode` for the extension requests. The *vncconfig* program can send nine types of extension requests: `X_VncExtSetParam`, `X_VncExtGetParam`, `X_VncExtGetParamDesc`, `X_VncExtListParams`, `X_VncExtSetServerCutText`, `X_VncExtGetClientCutText`, `X_VncExtSelectInput`, `X_VncExtConnect`, `X_VncExtGetQueryConnect`, and `X_VncExtApproveConnect`. For example, if a user types in the following command.

```
vncconfig -get passwordFile
```

A VNC extension request consisting of the `major_opcode` for the VNC extension, the `X_VncExtGetParam` type, and the parameter name "*passwordFile"* will be send to the X server. Based on the `major_opcode`, the VNC extension dispatcher in the *vnc.so* module will be called, and the dispatcher will process the request based on the request type. For the `X_VncExtGetParam` type, the password filename (mostly *"/root/.vnc/passwd"*) will be retrieved and sent back using a reply message. When *vncconfig* runs with no options, it is to support clipboard transfer to and from the VNC viewers. Without it, no clipboard support is provided.

Clipboard Transfer

When the *vncconfig* program is started with no options, it creates a simple window with three check boxes: "Accept clipboard from viewer", "Send clipboard to viewer", and "Send primary selection to viewer". When the VNC server receives `msgTypeClientCutText` from a VNC viewer, it calls *vncClientCutText* in the *clientCutText* function of *XserverDesktop* that sends a `VncExtClientCutTextNotify` event to the client *vncconfig*. At receiving the event, *vncconfig* sends a request of type `X_VncExtGetClientCutText` in the *XVncExtGetClientCutText* function to retrieve the client cut text. It then saves the retrieved text in the cut buffer using *XStoreBytes*, and declares itself as the owner of the clipboard and the primary selection.

When the VNC server detects a selection change, it calls the *SendSelectionChangeEvent* function that sends a `VncExtSelectionChangeNotify` event to the client *vncconfig*. At receiving the event, *vncconfig* calls *XConvertSelection* to request the server cut text from the owner. Then the *sectionNotify* function of requestor will be called. The *sectionNotify* function calls the *XVncExtSetServerCutText* function to

send the server cut text using a request of type `X_VncExtSetServerCutText`. The VNC server will send the server cut text to the viewers in the *serverCutText* function of *XserverDesktop*.

Mouse and Keyboard Events

When the VNC server receives `msgTypePointerEvent (or msgTypeKeyEvent)` from a VNC viewer, it calls the *pointerEvent* (or *keyEvent*) function of *Xserverdesktop,* in which the *LookupPointerDevice* (or *LookupKeyboardDevice*) will be called to find the pointer to the corresponding device, and the event will be delivered using its *processInputProc* function.

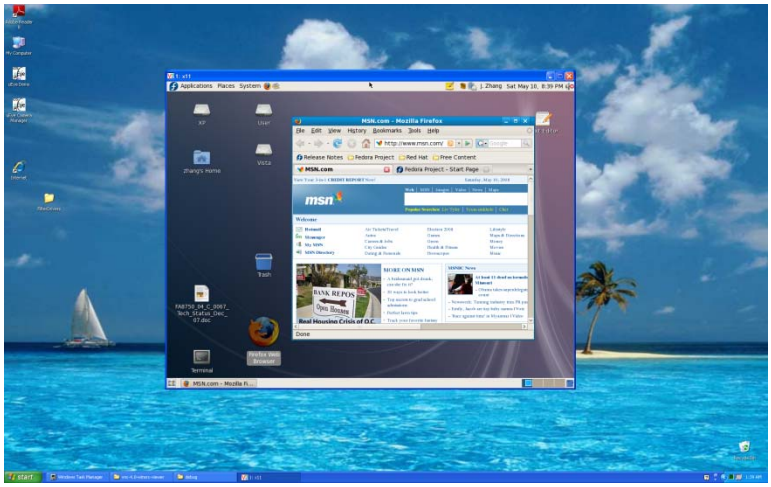

Figure 2.6 An X desktop on a Windows platform.

Figure 2.6 illustrates an X desktop on a Windows platform. The VNC for Unix is not suitable for application sharing. As Figure 2.5 shows, VNC is installed along with the X server. It is very difficult for the X server to get any information about an application which is started at the X client side. In fact, as described previously, in order to achieve clipboard sharing, an application called *vncconfig* must be running to assist.

3. Mirror-driver-based Approach

The VNC-based approach has two shortcomings. First, it needs to access the frame buffer located on the video memory, which is slow. Second, the *msgTypeFramebufferUpdate* message will be sent to the client only after the client requests it. To overcome these shortcomings, we introduce the mirror-driver-based approach. In the mirror driver, we keep a copy of the frame buffer (i.e. the mirror) in system memory. Second, we send the *msgTypeFramebufferUpdate* message to the client as soon as possible without waiting for a request from the client.

3.1 Mirror Driver

A mirror driver is able to intercept all the drawing commands issued from applications. Whenever an application calls a Win32 GDI (Graphics Device Interface) function, the GDI Graphics Engine will convert it into a graphics DDI (Device Driver Interface) function to be executed by the display driver. The GDI Graphics Engine, the display driver, and the mirror drive reside in the kernel of the operating system. Like the actual display driver, a mirror driver will receive all the graphics DDI functions generated by the GDI Graphics Engine. Our idea is to transport all the intercepted commands to another computer, the DataWall computer in our case, and to execute the drawing commands on the DataWall. Since the mirror driver is in the kernel mode, and it is not able to transport all the intercepted commands directly to the DataWall. That is why an application called *forwarder* is designed to retrieve and forward the intercepted commands. The application running on the DataWall to execute the drawing commands is called the executor. Figure 3.1 illustrates the architecture of the mirror-driver-based system.

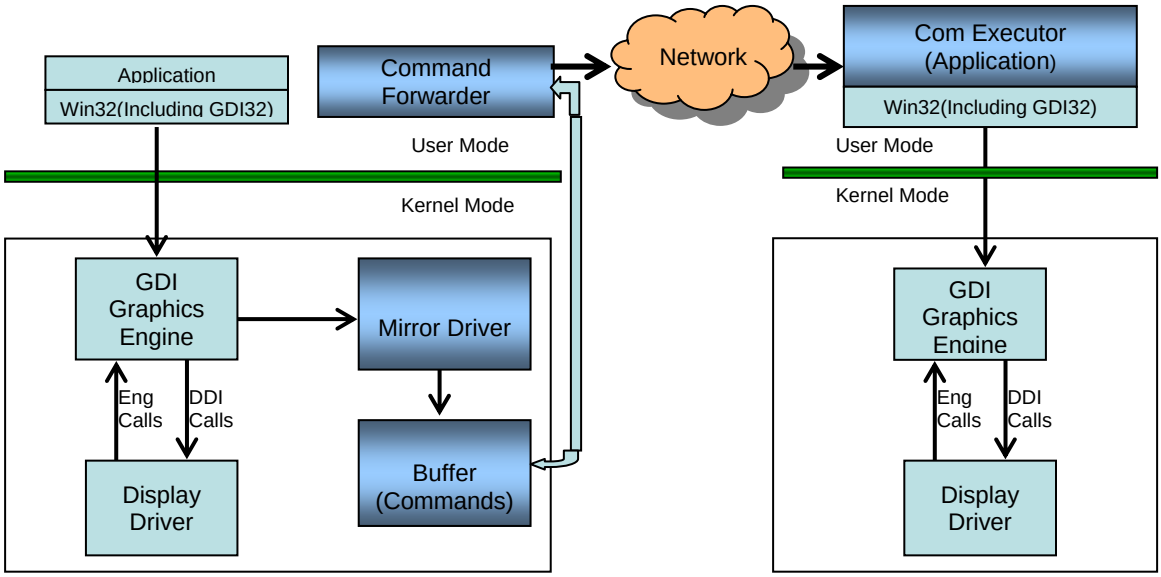

Figure 3.1 The architecture of the mirror-driver-based system.

However, the DDI functions to be executed have a different set of parameters from the GDI functions and some of the parameters are pointers (therefore context-dependent). For example, when an application draws a line from the current position to a specified point using the *Lineto* GDI function in Win32 API, three parameters, a handle to the destination device context, the x-and y-coordinates of the ending point are specified. The GDI Graphics Engine converts *Lineto* into the 9 parameter DDI function named

Drvlineto. When the executor receives the *Drvlineto* function, it has to convert it back to the *Lineto* function so it can be called by the executor. Some parameters are context dependent and cannot be sent directly. Therefore, we decided to maintain an engine-managed surface and to punt the drawing operations back to GDI to let GDI perform the drawing operations on the engine-managed surface. After each drawing, we extract the part of surface that is affected by the drawing operation and send the affected part to the DataWall.

To make the mirror driver functional, we only need to implement four required drawing functions, DrvCopyBits, DrvBitBlt, DrvTextOut, and DrvStrokePath. These functions ask the driver to perform the corresponding drawing operations on the device-managed surface. Due to the complexity of these functions, we will punt them back to the GDI engine by calling EngCopyBits, EngBitBlt, EngTextOut or EngStrokePath respectively with an engine-managed surface. In the DrvEnableSurface function, we create a device-managed surface using EngCreateDeviceSurface and create an engine-managed surface using EngCreateBitmap, and associate the engine-managed surface with the device-managed surface. When a DDI drawing function, such as DrvBitBlt, gets called with the device-managed surface, we get the engine-managed surface from the device-managed surface, and then pass the engine-managed surface as the destination surface of EngBitBlt. In this way, the engine-managed surface mirrors the actual screen and we know all the changes made to the screen.

Next we will discuss which part of the screen has been changed by each required DDI drawing function. DrvCopyBits has six parameters: psoDst, psoSrc, pco, pxlo, prclDest, and pptlSrc. We are interested in two parameters, psoDst and prclDest. psoDst is a pointer to a SURFOBJ structure that identifies the device-managed surface from which the engine-managed surface can be obtained. The prclDest is a pointer to a RECL structure that defines the area to be modified. The RECL structure contains four members which identify the top-left corner and the bottom-right corner of a rectangular area. Only the part of the screen within the rectangle identified by prclDest is changed and needs to be transferred to the DataWall. DrvBitBlt has eleven parameters: psoTrg, psoSrc, psoMask, pco, pxlo, prclTrg, pptlSrc, pptlMask, pbo, pptlBrush, and rop4. Similar to DrvCopyBits, we are only interested in two of them, psoTrg and prclTrg. psoTrg points to the device-managed surface, and prclTrg points to a RECL structure that defines the rectangular area to be changed.

DrvTextOut has ten parameters: pso, pstro, pfo, pco, prclExtra, prclOpaque, pboFore, pboOpaque, pptlOrg, and mix. Only two of them are needed. One is pso that points to the device-managed surface. The other is pstro that points to a STROBJ structure that defines the glyphs to be rendered and the positions at which to place them. The member, rclBKGround, in the STROBJ structure is a RECL structure that describes the bounding box for the string. The bounding box contains the area that may be affected by the glyphs. Therefore it is sufficient to transfer all the pixels within the bounding box to the DataWall.

DrvStrokePath is a little complex and needs more effort. It has eight parameters: pso, ppo, pco, pxo, pbo, pptlBrushOrg, plineattrs, and mix. Again, pso points to the device-managed surface from which we find the engine-managed surface. Pco points to a CLIPOBJ structure. All the lines and/or curves in the path may be enumerated preclipped in the CLIPOBJ structure. That is, all lines and/or curves fall in the clip region. Next we will talk about how to get the affected area from the CLIPOBJ structure. CLIPOBJ structure consists of six members: iUniq, reclBounds, iDComplexity, iFComplexity, iMode and fjOptions. iDcomplexity specifies the complexity of the region relevant to the present drawing operation. iDComplexity must be one of following values, DC_RECT, DC_TRIVAL or DC_COMPLEX. If iDComplexity is DC_RECT or DC_TRIVAL, the member rcclBounds reveals the rectangle in which the lines or curves will be bounded. If iDComplexity is DC_COMPLEX, pco contains a batch of rectangles that bound the lines and/or curves. In this case, we have to enumerate the rectangles. The CLIPOBJ_cEnumStart function can be called to determine the exact number of rectangles in the region and the CLIPOBJ_bEnum function enumerates the rectangles from the specified clip region. We union all the rectangles, and send all the pixels within the union to the DataWall.

3.2 Forwarder and Executor

The forwarder is responsible for retrieving the changes in the mirror driver. To retrieve the changes in the mirror driver, we implemented the *DrvEscape* function for communication between the forwarder and the mirror driver. We implemented five commands within the *DrvEscape* function: *GET_RESOLUTION*, *START_MIRROR_DRV*, *STOP_MIRROR_DRV*, *CHECK_CHANGE*, and *GET_CHANGE*. *GET_RESOLUTION* is to get the resolution and the color depth of the mirror device. Although it may not be necessary because the resolution is set by the application using *ChangeDisplaySettingsEx*, we keep this command there for convenience or for another application just in case. *START_MIRROR_DRV* is to ask the mirror driver to start drawing the mirror image on the engine-managed surface, and *STOP_MIRROR_DRV* is to ask to stop the drawing. Once the mirror driver is installed, all the functions for drawing the mirror images will be called. If the *START_MIRROR_DRV* command is not given, the called functions will return immediately without any actual drawing. After the *START_MIRROR_DRV* command, the drawing commands executed by the actual graphics board will be executed on engine-managed surface. Since the drawing in the mirror driver can be started and stopped for efficiency, it is recommended that the application immediately issue a whole screen *bitblt* operation after the *START_MIRROR_DRV* command. In this way, engine-managed surface will have the same content as the actual frame buffer. *CHECK_CHANGE* is to check whether any change is made to engine-managed surface. If the *CHECK_CHANGE* indicates changes have been made to the engine-managed surface, the *GET_CHANGE* command can be issued to retrieve the changes.

The executor on the DataWall creates an individual thread for each remote machine. The thread establishes a socket to communicate with the corresponding remote machine. Through the established socket, the thread first obtains the screen resolution and color depth of the remote machine. Based on the received screen resolution, the thread creates a window of the same size as the resolution. Each window has its own distinct color as its border so the users can distinguish the projections of the remote machines on the DataWall by their border colors. The window class is defined as follows.

```
class DWWnd{
public:
      DWWnd();
      HWND getHandle() const {return m_pWnd;}
      BOOL Create(const char name[], unsigned int style, unsigned int
            ex_style, int sx, int sy, int x, int y, int w, int h);
      BOOL Move(int sx, int sy, BOOL bRepaint);
      BOOL InBorder(int x, int y);
      void Destroy();
      LRESULT CALLBACK processMessage(UINT msg, WPARAM wParam,
                                                LPARAM lParam);

      HWND m_pWnd;
      int LEFT, TOP;
      int WIDTH, HEIGHT;
      char * SCREEN_BUF;
      char * buf;
      int screen_buf_lDelta;
      BITMAPINFO RGB32BitsBITMAPINFO;
      int DstX, DstY, Cur_Width, Cur_Height, SrcX, SrcY;
      int COLOR_DEPTH;
      int TODOWN_BITMAP;
      HBITMAP DirectBitmap;
      int border;
      BOOL drawingBorder;
      COLORREF BorderColor;
      HBRUSH hbr;

};
```

Variables LEFT, TOP, WIDTH and HEIGHT in the class DWWnd are the geometry of the window created on the DataWall for a remote machine. SCREEN_BUF holds the desktop image of the remote machine. Variable drawingBorder indicates whether a border for the window is needed. If the border is needed, border and BorderColor are the border width and the border color respectively. The processMessage function processes the messages sent to the window, such as WM_PAINT, WM_MOUSEMOVE, WM_WM_LBUTTONDOWN, etc. Whenever the executor on the DataWall receives an image update from a remote machine, a WM_PAINT message will be generated.

Figure 3.2 shows the flow of the image update from the remote machine to the DataWall. In the mirror driver running on the remote machine, a buffer called the mirror

driver buffer which is an engine-managed surface holds the current desktop image of the remote machine. The forwarder only sends the image update to the executor. For example, in Figure 3.2, three rectangles of images, shown in green, blue and light blue, have been updated on the remote machine. These images might be overlapped as shown in the figure. The forwarder computes a minimum bounding box to contain all the image updates. In this example, the minimum bounding box is marked as A in Figure 3.2. The forwarder sends the rectangle A of pixels to the executor. For each remote machine, the executor on the DataWall keeps a window buffer whose size is the same as that of the mirror driver buffer on the remote machine. When the executor receives the rectangle A, it will update the window buffer by placing the rectangle A at the corresponding position. The window buffer will be consistent with the mirror driver buffer when the updating was completed. The window buffer is reflected in the window on the DataWall. When the window buffer is updated, the executor called the InvalidateRect function to trigger a WM_PAINT message to be sent to the window. The processMessage function paints the rectangle A with the corresponding image update.

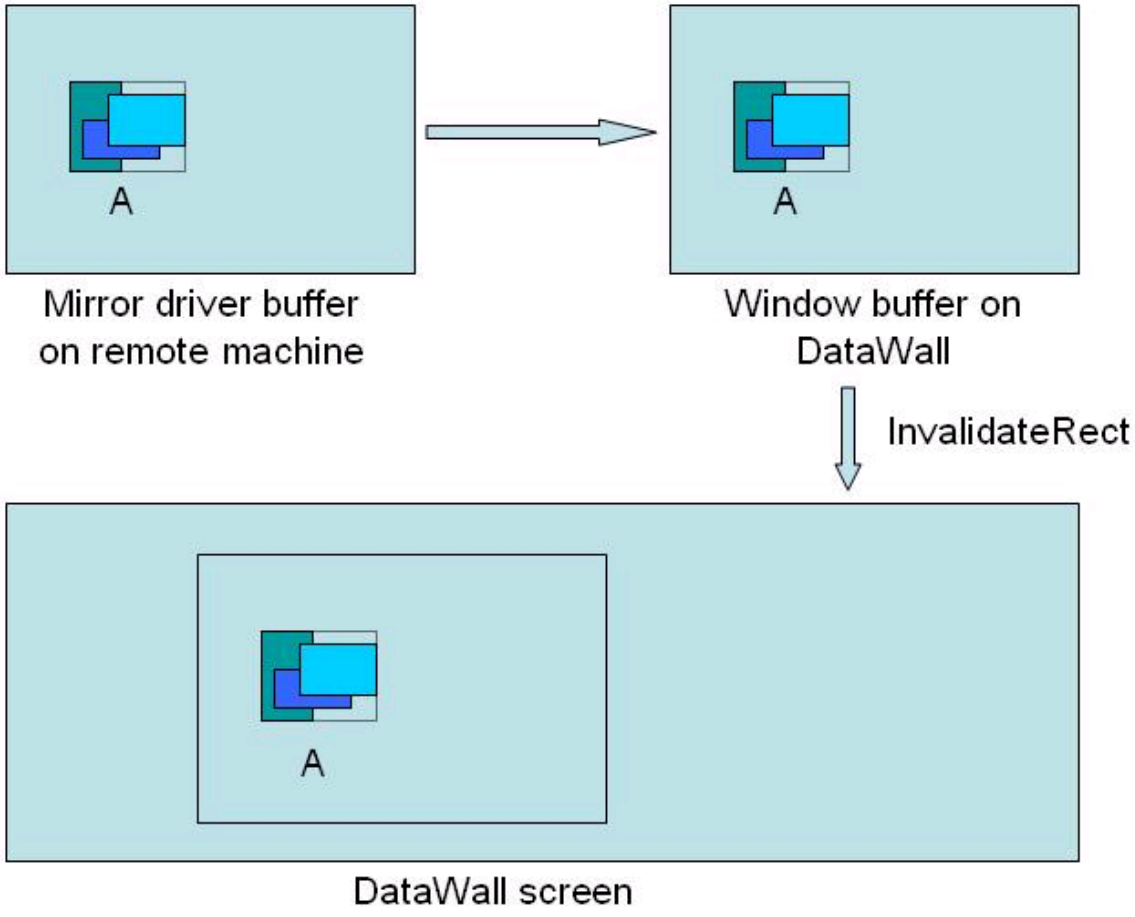

Figure 3.2 The flow of the image update from the remote machine to the DataWall.

The executor is also responsible for input event collection. The processMessage function will capture and send the mouse and keyboard events to the remote machine. The forwarder on the remote machine will simulate these events locally to control the remote machine. Application sharing is also implemented as in the VNC-based approach.

3.3. Reverse Use of Mirror Driver

Usually the mirror driver and the forwarder run on an individual computer called an application computer, and the executor runs on the DataWall as shown on the left side of Figure 3.3. The reverse use of the mirror driver is to run the mirror drive and the forwarder on the DataWall, and to run the executor on an individual computer, referred to as a control computer as shown on the right side of Figure 3.3. The job of the executor is to display the content of its remote partner (i.e. the mirror driver and forwarder) and to provide input events to the partner. When the executor runs on the control computer, its partner (the mirror driver and forwarder) will run on the DataWall computer, and their main functionality is to let the executor to control a part of the DataWall. This is particularly useful when a user is interested in an application on the DataWall and wants to interact with it. Only one instance of the forwarder needs to be run on the DataWall. One forwarder on the DataWall can serve multiple executors (three in Figure 3.3).

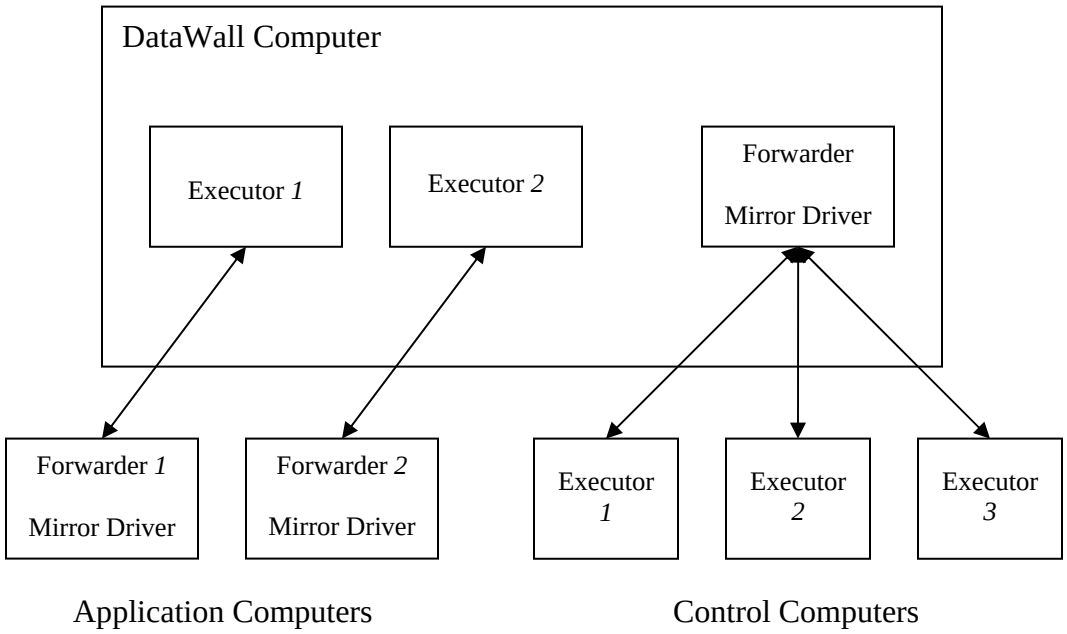

Figure 3.3 Executor can run on the DataWall or a control computer.

The executor always keeps a buffer (referred to as the window buffer in previous section) of the same size as the frame buffer size of its partner (the mirror driver and forwarder) regardless of the computer on which the executor runs or the mode in which

the executor runs. When the executor runs on the DataWall computer, its main job is to project an application from the corresponding application computer to the DataWall. In this case, a main window whose client area has the exactly same size as the buffer will be created. The client area reflects the frame buffer (i.e. the desktop) of the forwarder. Therefore, one-to-one ratio is kept between the main window and the application computer desktop. Since the application sharing is interested in this case, the main window will be hidden. However, the windows that belong to the application to be shared need to be displayed. To display these application windows, the same numbers of windows will be created and these windows will be positioned at the same locations within the hidden main window. Figuratively, to display the windows that belong to the application to be shared, we push up these windows from the hidden main window to be seen. Since multiple instances of the executor can run on the DataWall, each application window will be bounded by a different color to indicate a different source application computer.

When the executor runs on the control computer (and its forwarder runs on the DataWall computer), the desktop sharing is of interest, and its main job is to let the user see the DataWall clearly on the local computer (i.e. the control computer) and interact with the DataWall directly from the control computer. The executor in this case can run in windowed mode or in full-screen mode, and a user can switch between these two modes by pressing "F8" to as shown in Figure 3.4.

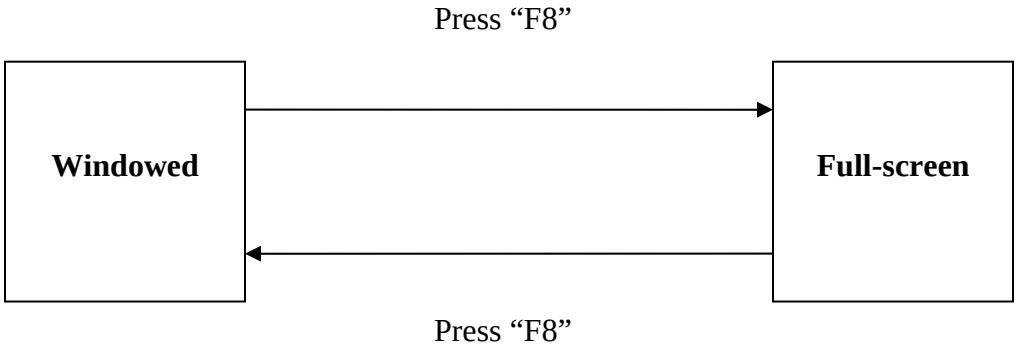

Figure 3.4 Switch between the windowed and full-screen modes.

In the windowed mode, a top-level window will be created to show the DataWall. In this mode, one-to-one ratio is kept between the created window and the DataWall. Since the created window or even the whole control computer screen is smaller than the DataWall, only a part of the DataWall can be seen at one time and horizontal and/or vertical scroll bars are added to let the user to select the part to be seen. A user can use the scroll bars to select the interesting part of the DataWall. To interact with the DataWall, a user can use his/her local mouse and keyboard within the window. In the full-screen mode, no scroll bars will be shown. However, the DataWall can be shown at a scale-down ratio instead of one-to-one. The whole DataWall, a monitor of the DataWall

or a window of a DataWall can be shown at a scale-down ratio. The one-to-one ratio can still be shown. In the one-to-one case, scroll bars are still not shown, but a user can use the arrow keys to move the portion to be seen. In all the cases, the user can use his/her local mouse and keyboard to interact with the DataWall.

Under the full-screen mode, we use a topmost tool box, referred to as the state control box, to select one of seven states: green F, red M, green M, green W, red W, red 1-1 and green 1-1. F stands for “Full View”, M for “Monitor View”, W for “Window View”, and 1-1 for “1-1 View”. In a green mode, any input event will be sent to the DataWall. In a red state, a user can select a part (a monitor, a window, or a rectangular area of the same size as the local computer) of the DataWall for view. Four arrow buttons are added to the state control box. A user can use these arrow buttons in a green state to move the area of interest by a monitor, a window or a few pixels on the DataWall. Figure 3.5.a shows the red W state in which the whole DataWall is shown at a reduced size on the control computer, and a user can use the mouse to select a window to be operated. Once a window is selected, it enters the green W state, and a user can immediately provides input to the DataWall. If the user is not happy with the selected window, the user can either use the arrow buttons to choose a neighbor window or go back to the red W state by clicking the W button and then use the mouse to select another window. Figure 3.5.b shows the details of the state control box.

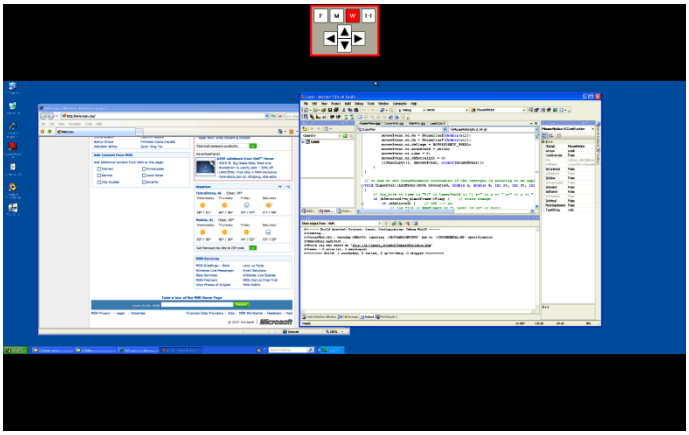

Figure 3.5.a The executor in the red W state.

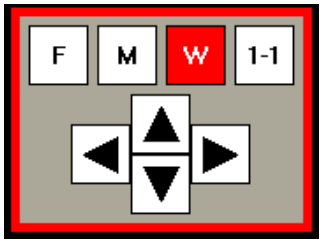

Figure 3.5.b The state control box.

To let multiple users to control the DataWall simultaneously, we also added a *WorkWithLaser* option to the forwarder. When *WorkWithLaser*=0, the executor works with the forwarder in both ways, that is, the forwarder sends the frame updates to the executor and the executor sends the input events to the forwarder. When *WorkWithLaser*=1, the executor still receives the frame updates from the forwarder, but the executor sends the input events to the Laser client. (Laser Client is to be discussed next seciton.) The Laser client has the ability to dispatch the input events to drive the system cursor of an application computer in addition to that of the DataWall computer. The forwarder sends the input events to drive the system cursor of the DataWall computer only.

With *WorkWithLaser*=1, the forwarder first tries to find the Laser client. If the Laser client exists, the forwarder will get the port number used by the Laser client for serving input clients, using the *SendMessage* function with Msg=WM_USER+98 and WParam=5, and then forwards the port number to the executor. The executor will use the port number to connect itself to the Laser client. Once connected to the Laser client, the forwarder sends an ADD_CURSOR request to ask the Laser client to allocate a software cursor as shown in Figure 3.6 to represent itself. (Of course, before the executor disconnects itself from the Laser client, it sends a DEL_CURSOR request to deallocate the software cursor.)

Figure 3.6 The software cursors managed by the Laser client.

The executor will monitor the following window messages: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_RBUTTONDOWN, WM_RBUTTONUP, WM_MOUSEMOVE, WM_MOUSEWHEEL, WM_KEYDOWN, and WM_KEYUP. For each message, the executor will fill the following `InputEvent` structure as shown below and sent it to the Laser client using an INPUT_DATA messge type. The Laser client will process the INPUT_DATA messge type similarly to the processing of those from laser pointers. At receiving an INPUT_DATA messge, the Laser client will dispatch the message to an application computer or simulate it on the DataWall computer. Furthermore, we now simulate the WM_MOUSEMOVE event even if no mouse button is down. Therefore we can see some window effects such as tooltips.

```
typedef struct _input_event {
      int eventtype;    //mouse 1; key 2.
      WORD character;
```

```
        int state;          // move 1; press 2; lift 4; or delta.
        int button;         // nothing down 0; left 1; right 2; sgl clk 3;
                            // dbl clk 4; middle 5; wheel 6.
        int x;
        int y;
} InputEvent;
```

4. Laser Pointers as Input Devices to DataWall

A laser pointer is usually used by a presenter to direct audience's attention. It is also natural to think of using a laser pointer as an input device to a computer with a large display such as the DataWall. In fact, the movement of the laser dot maps naturally to the cursor movement, and the push and release operations on the laser pointer map naturally to the button-down and button-up operations on the mouse. In a previous project, we developed Laser software that used multiple analog security cameras to track laser pointers. However, image processing is very computation-intensive. In this project, we divided the Laser software into Laser server and Laser client. The Laser server is responsible for computation-intensive image processing and can be run on a different computer from the Laser client. Furthermore, we used digital cameras in place of analog security cameras.

4.1 States of Laser Software

In the original Laser software, there are 5 global stages: INIT(0), CONFIGURED(1), CALIBRATED(2), STARTED(3) and STOPPED(4). Although we used the term *stage* in the original software, we prefer the term *state* in the new version. Now the states of INIT, CONFIGURED and CALIBRATED are maintained in the server for each monitor as well as each client. A client can drive multiple monitors (or screens). A client is in a state of *x* (INIT, CONFIGURED or CALIBRATED) if all of its monitors are in the state of *x*. The states of STARTED and STOPPED are still kept as global for the server itself. When the server is executed, it first checks the number of monitors available on the sever computer, then reads the configuration information for each client that was saved in *laser_server.ini* in the previous execution, and finally listens on a port that has a default value of 10002. The configuration information for each client is permanently identified by its IP address. Even when there is no connection established from a client, a user can check any one of three cameras and display the captured images on the selected server monitor. When a client is executed on a computer, it will establish a connection to the server. Once a connection is established, the client can be also identified temporally by the socket. The server will first ask the client to send its monitor configuration information. If the server does not have any information about the client, an entry is created for the client and the client and its monitors are in the INIT state. If there is an entry for the client, there are three cases. If the monitor configuration information matches the received information, the client keeps the same state. If the number of monitors does not match, the client goes back to the INIT state. If the number of monitors matches, but the resolution does not match, the client goes back to the CONFIGURED

state. When a client goes back to the INIT or CONFIGURED state, all of its monitors also go back to the INIT or CONFIGURED state respectively.

Configuration is to associate the cameras with the screens. Configuration is initiated by the user at the server side. A user can also check a camera during configuration because one of the reasons for camera check is to make sure the association between a camera and a screen is correct. When the camera check command is issued by the user, the drawing command is sent from the server to the corresponding client. The client performs the drawing operation on the corresponding screen, and sends an acknowledgement to the server. On receiving the acknowledgement, the server starts to continuously grab the images and display them live on a designated server monitor. With the help of live capture, the user can also adjust the camera to make sure the screen is completely captured by the camera, and the camera iris setting is appropriate. Once a screen/monitor is configured with a camera, the configuration information is saved into the laser.ini file. Once all the monitors of a client are configured, the client is considered as configured, i.e. in the CONFIGURED state.

Calibration is to establish the relationship between the pixels in the camera and the pixels in the corresponding screen. The calibration has to be done one by one. Once a camera is chosen and the collaboration command is issued at the server side, the calibration process starts. The server takes the active role in the calibration process. The server issues the pattern-drawing command to be sent to the corresponding client. The client will draw the pattern on the corresponding screen, and sends an acknowledgement to the server. The server then takes a snapshot of the corresponding screen, analyzes the snapshot, and shows the result on a designated server monitor. If the calibration is successful, the user can accept the calibration and the collaboration information will be saved in laser.ini file to replace previous calibration information, if exists. Whenever all the monitors of a client are calibrated, the client is considered as calibrated, i.e. in the CALIBRATED state.

The server can enter the STARTED state if there are at least one client is in the CALIBRATED state. The server will detect the laser dots and transform the coordinates of the detected laser dots from the camera space to the screen space. The transformed dot positions will be sent to the corresponding client if the client is connected. If the client is not connected, the transformed dot positions will not be sent. However, when the server in the STARTED state, if a CALIBRATED client establishes a connection with the server and still maintains the CALIBRATED state, the transformed dot positions will start to be sent to the client. A server in the STARTED state can go into the STOPPED state at request of the user.

4.2 Communication between Laser Server and Client

When the Laser server is started, it first reads the configuration and calibration information from the *laser_server.ini* file, if exists. It then performs the Winsock TCP/IP

server procedure: *WSAStartup*, *socket*, *bind*, and *listen*. To combine the socket input with the user input, the server uses *WSAAsyncSelect* to ask for *MSG_WINSOCK* (a user-defined message) to be sent whenever there is a connection request from a client. When a client is started, it reads the server's IP address and the port number from *laser_client.ini*, and performs the connection procedure: *WSAStartup*, *socket* and *connect*. When a connection request is made from a client, the server will receive the *MSG_WINSOCK* message. When processing the *MSG_WINSOCK* message, the server accepts the connection, and just puts the accepted socket in the *NewLaserServers* list to be processed later. In this software, the accepted socket is blocking while the listening socket is non-blocking. Since the accepted socket inherits the non-blocking mode from the listening socket, we have to use *WSAAsyncSelect* and *ioctlsocket* to convert the mode from non-blocking to blocking.

In our design, we use a three-tier architecture: *NewLaserServers*, *CandDisplay*, and *ConfDisplay*. As mentioned in the previous paragraph, *NewLaserServers* contains the newly accepted sockets (or connections). From time to time, the server calls the *ProcessNewLaserServers* function to process these newly accepted connections. In *ProcessNewLaserServers*, the server asks the client to send its display information including the number of monitors in the display and the geometry of each monitor. Once the display information is received, the display at the client end of the connection becomes a candidate display to be kept in *CandDisplay*. A candidate display can be configured and become a member of *ConfDisplay*. When a display is configured, each of its monitors is associated with a camera. All members of *ConfDisplay* will be saved into *laser_server.ini* under the *DISPLAYS* section to be read whenever the server is restarted. Each monitor of a configured display can then be calibrated. If the calibration is successful, the calibration information will also be saved into the *laser_server.ini* file. Whenever all the monitors of a configured display are calibrated, the display is calibrated. Once a display is calibrated, the corresponding client is able to receive the laser dots from the server if a connection exists between the server and the client. Sometimes although the client's display is configured and calibrated, the client may not be connected to the server. This could happen when the server just starts up or when the connection is lost. Under this situation, when a connection is established from the client, the *ProcessNewLaserServers* function moves the client's display to *ConfDisplay* without going through *CandDisplay*, and the laser dots can be sent to the client immediately. Therefore if a client is stopped and restarted, the client can receive laser dots immediately. The server detects the lost of connection when sending or receiving its messages. From time to time, the server also calls the *ProcessClosedSockets()* function to check the connections. The *ProcessClosedSockets()* function uses the *LINK-TEST* message (to be described next) to verify the socket (or the connection) is still valid.

The Laser client is kind of passive in our design. After the connection is made to the server, the client creates a thread to wait for messages from the server. Currently, the following commands are used between the Laser server and the Laser client.

- `LINK_TEST`: to test whether the client and the server is still connected.

- `DISPLAY_INFO`: to send the client's display information to the server.
- `DRAWING_CMD`: to send a drawing command to the client. Currently there are three drawing commands: `BLANK_SCN` for drawing a blank screen, `SOLID_RECTS` for drawing a sequence of solid rectangles, and `SOLID_DOTS` for drawing a sequence of solid dots. If the drawing canvas does not exist at the client, the client will first create a full-screen canvas before drawing anything.
- `DRAWING_END`: to inform the client of the end of drawing. The client will destroy the canvas if it has not done so.
- `DRAWING_ACK`: to be sent by the client to inform the server that the drawing command has been executed.
- `LASER_DOTS`: to send the detected laser dots to the client.
- `STATE_SET`: The state is maintained by the server, and the server uses this command to set the state of the client to match its state.

The new Laser server can work with multiple clients, and one of the clients can be residing on the same machine as the server. When a client is residing on the same machine as the server, the server and the client will share the same display. In this case, the *DRAWING_CMD* message will not be sent. Instead, the server will perform the drawing directly on the shared display. However, the *LASER_DOTS* message is still sent through the socket. It is expected that all the clients will be connected to the server using a local area network. The test shows the performance is satisfactory with no obvious delay. Since all the messages in our design are small in size, *TCP_NODELAY* is set using *setsockopt* to disable the Nagle algorithm for send coalescing.

4.3 Digital Cameras for Tracking Laser Pointers

The new Laser software uses USB 2.0 digital cameras for laser dot detection. Specifically, the new Laser software uses three uEyeLE monochrome cameras of a resolution of 752x480 pixels each (model: UI-1225LE-M). Figure 4.1 shows the new camera system installed in a portable DataWall. The new camera system works with a DataWall of a resolution of 3840x1024 pixels now. For a higher resolution DataWall, we may need to use more cameras or higher resolution cameras.

Figure 4.1 The new camera system installed in a portable DataWall.

The hardware components in the new camera system include three UI-1225LE-M cameras (http://www.1stvision.com/cameras/camera_detail.php?id=150#UI-1225LE-M), three Tamron 13VM2812AS lenses (http://www.spytown.com/13vm2812as.html), and three USB 2.0 cables (http://www.cdw.com/shop/products/default.aspx?EDC=376238). There is only one cable (the USB cable) attached to a camera in the new system instead of three (sync, video, and power cables) in the previous system.

Because the new camera system involves multiple cameras, the software needs to create multiple threads to synchronize the capturing of multiple images, to detect the laser dots from the captured images, and to merge the detected laser dots. Figure 4.2 illustrates the interaction and synchronization among these different threads. The process starts with calling the *is_FreezeVideo* function to ask the three cameras to digitize the images. When a camera completes the digitization of an image, the camera sends an *IS_SET_EVENT_FRAME* event which will wake up the *Laser Dot Detection* thread responsible for detecting the laser dots in the image. The double buffer technique is used to store the digitized images; therefore the Laser Dot Detection thread can send the *Ready* signal to the *Trigger* thread before processing the image. In this way, the image digitization and the image processing can be done in parallel to increase the frame rate. When an image is processed, the Laser Dot Detection thread will let the *Laser Dot Merger* thread know the laser dots from its camera are ready. When the Laser Dot Merger thread receives all three sets of laser dots, it will eliminate the duplicates in the sets and merge these three sets into one. Concurrently, when the Trigger thread receives the Ready signals from all the three Laser Dot Detection threads, it issues new digitizing commands to the three cameras again, and the process repeats.

We tested the new camera system on a DELL OptiPlex 745 mini-tower. The 745 mini-tower has an Intel's ICH8 south bridge that consists of two on-board USB 2.0 controllers. When three cameras are connected to the same controller, we are able to set a pixel clock of 17MHz to each camera. With the 5ms exposure time, we are able to achieve a frame rate of 29.7 frames per second. When splitting the three cameras to the two controllers, we set a pixel clock of 25 MHz to all the three cameras and achieve a frame rate of 38.7 frames per second. The test result shows the three camera system will work when installed on any desktop or laptop with an on-board USB 2.0 controller.

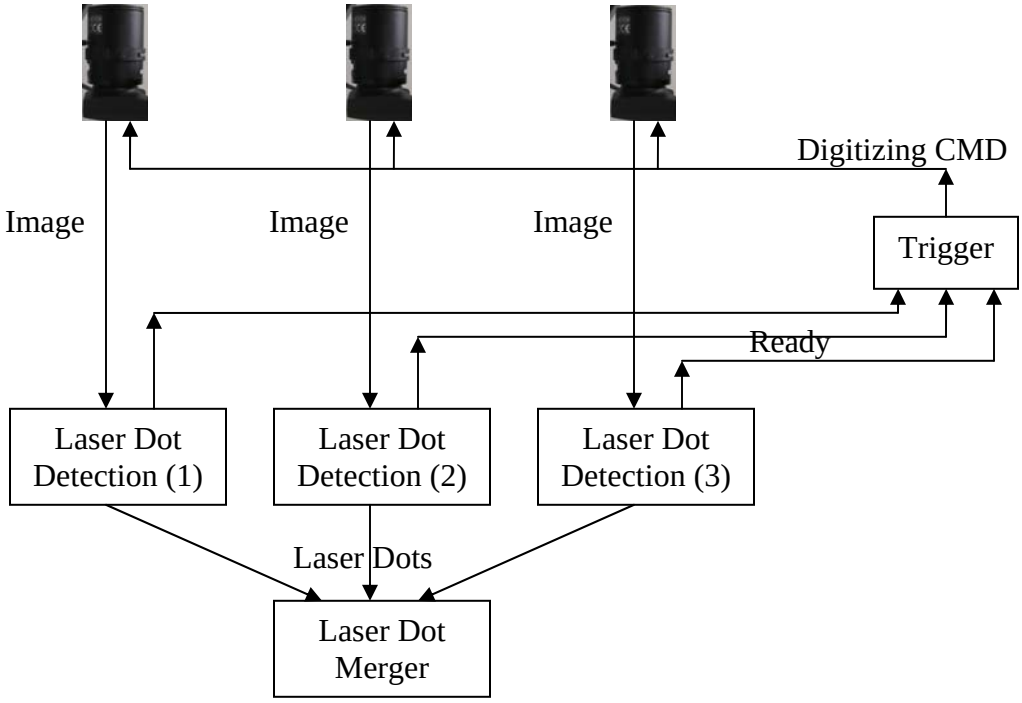

Figure 4.2 The interaction and synchronization among different threads.

4.4 User Interface for Laser Software

Once the Laser sever is started, a tray icon will appear on the task bar. The tray icon will have a pop-up menu as shown in Figure 4.3. It consists of *Camera Check*, *Auto Configuration*, *Camera Configuration* (manual), *Auto Calibration*, *Camera Calibration* (manual), and *Start Laser Server* menu items.

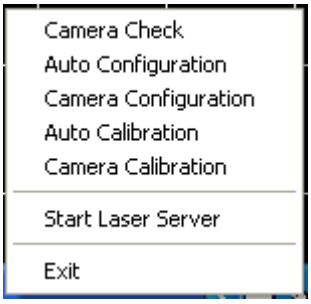

Figure 4.3 The pop-up menu for the Laser server.

The *Camera Check* allows the user to check whether the camera is properly positioned and to find the intensity difference between a laser dot and the white

background. If a camera is already configured to a monitor, the software will draw, on the corresponding monitor, a wire rectangle to indicate the boundary of the monitor with a solid square at each corner, as shown in Figure 4.4. If the camera is not configured, no rectangle or square will be drawn. However, the red wire rectangle that indicates the camera image size will always be drawn. One purpose of the *camera check* is to make sure the monitor image is positioned within the red rectangle. If the camera is configured, make sure the wire rectangle image which indicates the monitor boundary is positioned within the red wire rectangle. If the camera is not configured, the job is a little hard to put the monitor image within the red rectangle. It is recommended a three-step procedure be followed to properly position the camera. In the first step, use *Camera Check* to roughly aim the camera at the corresponding monitor. In the second step, perform *Auto Configuration*. Once the cameras are configured by *Auto Configuration*, use *Camera Check* again to make sure the image of the white wire rectangle is within the red wire rectangle. The second purpose of *Camera Check* is to find a threshold to separate the laser dot and the white background. That is why four solid squares are drawn at four corners. Usually the laser dot is very bright and has an intensity of 255. Therefore, the threshold is usually set at 255. If the white solid rectangles also show an intensity of 255, the camera iris has to be closed down to reduce the intensity of the white background.. The mouse and arrow keys can be used to move the small red-cross to inspect the pixel intensity within the image. The determined threshold is to be put into the *laser_server.ini* file manually.

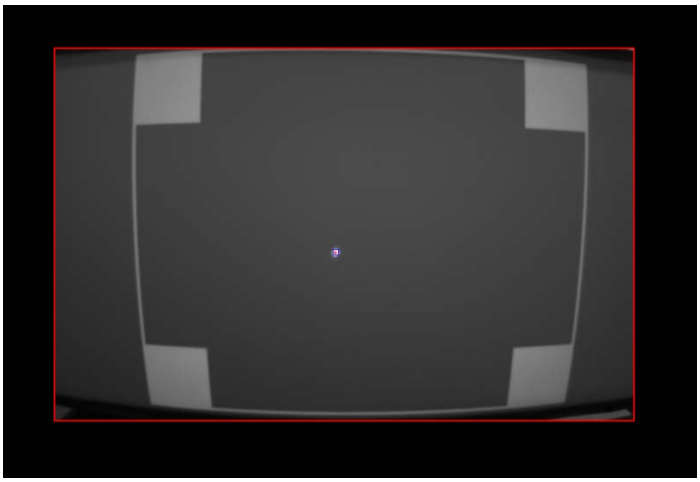

Figure 4.4 Camera check.

The *Auto Configuration* is to associate a camera with the monitor it is watching automatically. For each monitor, we draw one big solid circle and one small solid circle on it, and ask which camera can see both circles. If a camera can see both circles, the camera is associated with the corresponding monitor. In addition, the camera's orientation is determined by the relative location of the big and small circles within the image. Once the *Auto configuration* is done, its result can be verified using the *Camera*

configuration as shown in Figure 4.5. Of course, the *Camera configuration* can be used to manually configure the cameras as the name implies.

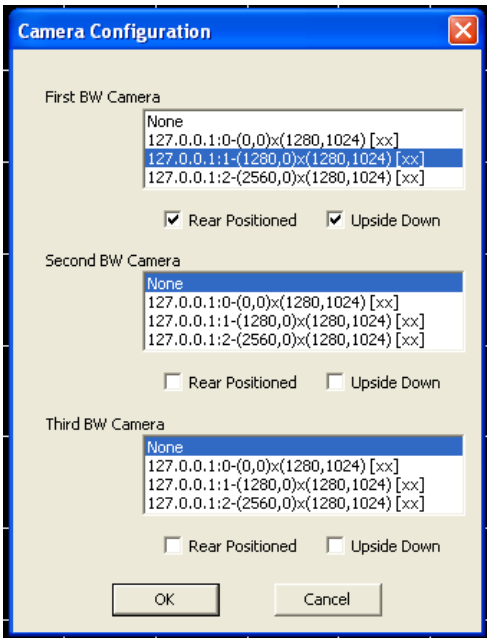

Figure 4.5 Camera configuration.

After the cameras are configured, the *Auto Calibration* can be used to calibrate all the configured cameras. For a configured camera, if you see a dot array with green coordinates as shown in Figure 4.6, the calibration of that camera is successful. Otherwise, a manual calibration may be needed. At camera calibration, the intensity difference between the white dots and the black background is important. Again you can use the mouse and arrow keys to move the small red-cross to inspect the pixel intensity within the image. Once the cameras are calibrated, the server can start the laser dot detection and distribution after the *Start Laser Server* menu is clicked.

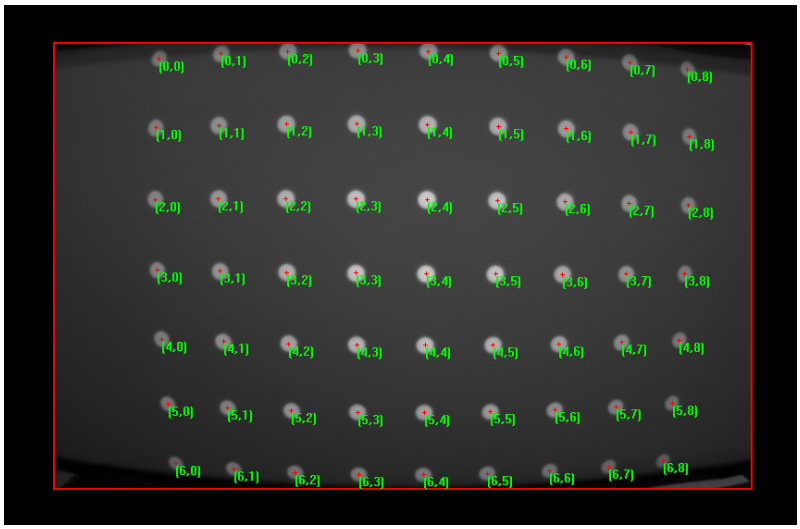

Figure 4.6 The image after a successful camera calibration.

4.5 Soft Keyboard for Text Entry

With the help of a MRW (Mouse Resource Window) as shown in the top part of Figure 4.7, a laser pointer can act just like a mouse. However, a user is not able to type in even a single line of text such as a URL (Uniform Resource Locator) into a web browser. To address this issue, we added a soft keyboard, shown in the bottom part of Figure 4.7, for use with a laser pointer.

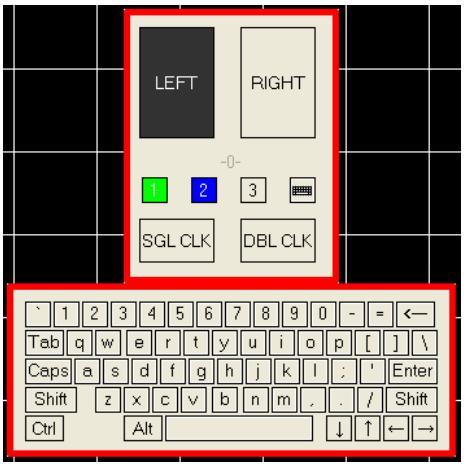

Figure 4.7 A soft keyboard for use with a laser pointer.

To save the DataWall space, the soft keyboard will not show by default. To show the keyboard, click the keyboard icon in the MRW with a laser pointer, and to hide the

keyboard, click the icon again. The starting position for the keyboard when it is shown is just below the MRW. To move the keyboard to another location, a user can use the laser pointer to click a non-key position and drag the keyboard to the desired location. To type in a character, a user needs to click the laser pointer at the corresponding key position.

The soft keyboard performs similarly to the soft keyboard provided with a PDA for use with a stylus. There are three kinds of keys on the soft keyboard: regular keys such as letters or digits, permanent state change keys such as *Caps*, and temporary state change keys such as *Shift*, *Ctrl* or *Alt*. When a temporary or permanent state change key is clicked, the new key state will be recorded, and the keyboard will reflect the state change. The state change can be reversed by clicking the same key again. When a regular key is clicked, the virtual key codes for these recorded key states and the regular key will be sent to the corresponding application. The temporary key states will be cleared after a regular key is clicked and processed.

5. Multiple Computer Interaction and Annotation

A mouse is usually limited within the computer to which it is attached. If you have two computers on the desktop, you need two sets of mice and keyboards to interact with these two computers. In this project, we define a virtual computer as consisting of multiple physical computers, and use the same set of mouse and keyboard to interact with all the physical computers in a virtual computer. To achieve this, we developed a mouse filter driver as well as a keyboard filter driver. A mouse filter driver can be installed with a mouse. Once the filter driver is installed, we can intercept all the mouse events generated from the mouse. We also represent the mouse with a software cursor, and use the intercepted mouse events to drive the software cursor. With the help of socket communication between computers, we are able to move the software cursor between computers. In this way, the software cursor can interact with multiple computers. With the mouse filter driver, we also developed annotation by multiple users across multiple computers. Figure 5.1 illustrates how the mouse filter driver is used for interaction and annotation within a virtual computer consisting of two physical computers.

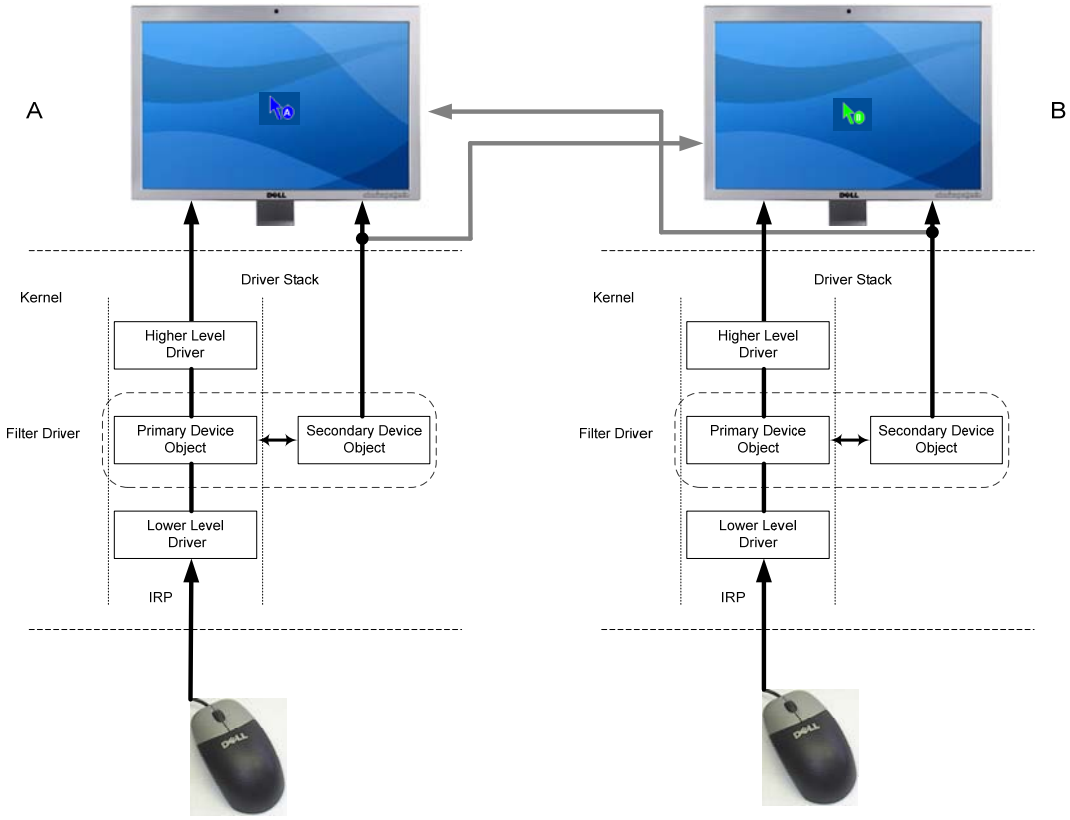

Figure 5.1 Mouse filter driver for multiple computer interaction and annotation.

5.1 Mouse Filter Driver

A filter driver for an input device is the best way to capture input events and pass them to an application on a Windows platform. Usually an application sends custom device I/O control requests (IOCTLs) to an added filter driver to retrieve the input events. The filter driver is able to get the custom requests as long as it is the upper-most filter in the stack, because that is the one that handles the request first. If another filter driver is added on top of it, the custom requests could be rejected by the filter above it. To address this issue, we create a standalone named device object, called control device object, to handle communication with the application. For an application to access the named device object, we need to create a user-visible interface either by a symbolic link to the device object name or a unique GUID. The application can open the standalone device object via the symbolic link name or the unique GUID with the CreateFile function and use the DeviceIoControl function to issue custom IOCTLs that will be sent directly to the control device object. Figure 5.2 shows the filter driver with two device objects. Device object 1 (unnamed) in the figure is inserted in the driver stack. It filters IRPs from up and captures input events from bottom. Device object 2 (named) reads the captured input events captured by device object 1, and sends them out of kernel to an application. There is a tool for us to view all the active device objects associated with drivers in Windows. Figure 5.3 is a screen shot of the installed mouse filter driver with two device objects.

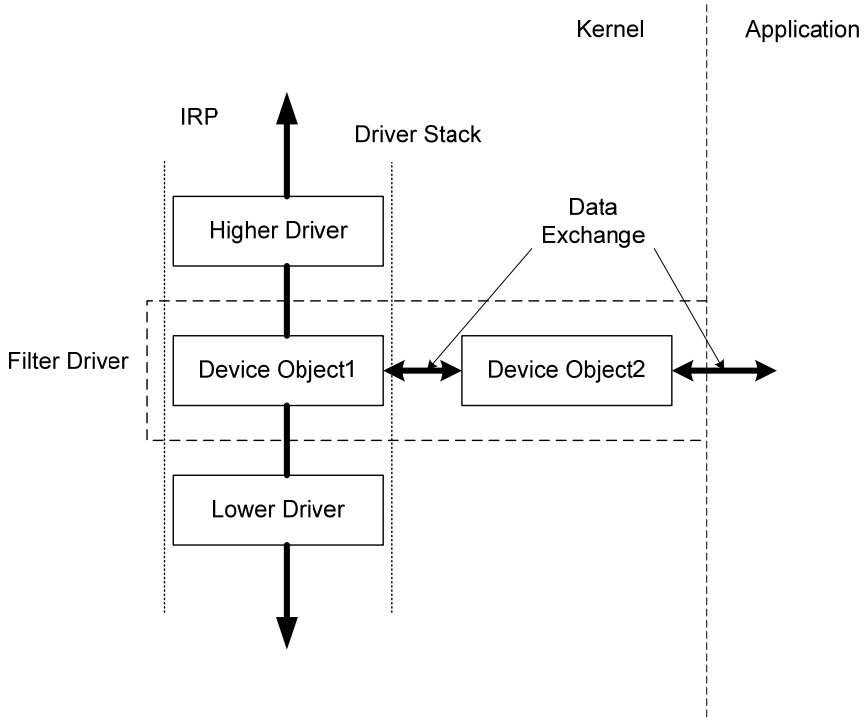

Figure 5.2 The filter driver with two device objects.

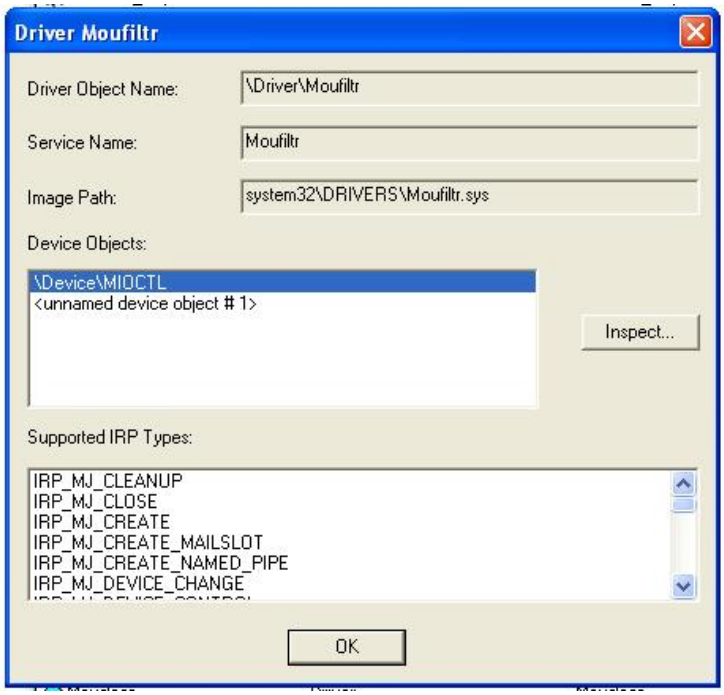

Figure 5.3 A screen shot of the installed mouse filter driver with two device objects.

5.2 Mouse Event Acquisition

The API function for a user-mode application to send an I/O control request to the filter driver is DeviceIoControl. The application first creates a handle to the device by calling CreateFile function in the following manner.

```
Handle = CreateFile("\\\\.\\MIoctl", GENERIC_READ | GENERIC_WRITE,
                                0, NULL, OPEN_EXISTING, flags, NULL);
```

Then, the application uses DeviceIoControl to exchange data in and out of the kernel with InputData and OutputData buffers as follows.

```
DeviceIoControl(Handle, Code, InputBuffer, InputLength,
                    OutputBuffer, OutputLength, &BytesReturned, NULL);
```

The Code argument to DeviceIoControl is a control code that indicates the control operation to be performed and how the input buffer and output buffers can be accessed in the kernel. That is a 32-bit numeric constant (DWORD) that can be defined using the CTL_CODE preprocessor macro that is part of both the DDK and the Platform SDK. Both the application and the driver should have the code definition in their header files. The least significant 2-bits in the code (i.e. the third argument to CTL_CODE) indicate the buffering method that tells how the filter driver can access the input buffer and output buffers supplied by the application. There are four possible valid values: METHOD_BUFFERED, METHOD_OUT_DIRECT, METHOD_IN_DIRECT, METHOD_NEITHER. The sample IOCTL in DDK under the directory of src\general\ioctl demonstrates how to use these four different types. Currently we are using METHOD_BUFFERED. With METHOD_BUFFERED, the I/O Manager allocates a kernel buffer of the size that is the maximum of InputLength and OutputLength for both input and output. Before the corresponding major function is called, the data in InputBuffer is copied into the allocated kernel buffer. Before the major function is returned, we put whatever we want to send back to the application, i.e. the captured mouse input events, in the allocated kernel buffer. The I/O manager will copy the data from the kernel buffer to OutputBufer after the major function returns. The number of bytes put in the kernel buffer needs to be indicated in the IoStatus.Information field of the IRP which is in turn put in the BytesReturned variable by the I/O manager. That is how an application uses the DeviceIoControl to get the data into and out of the filter driver.

For efficiency, we allow the application to define the buffer size to be retrieved from the filter driver with the I/O control code IOCTL_SET_BUFSIZE. To set the buffer size to be read from the filter driver, we use this function with the `size` value.

```
DeviceIoControl ( handle, (DWORD)IOCTL_SET_BUFSIZE,
   &size, sizeof(int), NULL, 0, &bytesReturned, NULL);
```

The keyboard filter driver is designed with the same principle as that used in the mouse filter driver. No detail is to be given here.

5.3 Interaction and Annotation

In our system, every participant machine could have an option to use a set of mouse and keyboard with their filter drivers installed. If the option is off, this machine will only be controlled by the other participants. If the option is on, the system creates a software cursor. The software cursor is able to go beyond its own physical computer and interact with any computer within the virtual computer, which means the software cursor from machine A could be on machine B and do whatever B's system cursor does.

The mouse and keyboard data packages obtained from filter drivers need to be converted when we use the SendInput function to synthesize keystrokes, mouse motions, and button clicks. The data structure obtained from the mouse filter driver is:

```
typedef struct MOUSE_INPUT_DATA {
  USHORT  UnitId;
  USHORT  Flags;
  union {
    ULONG  Buttons;
      struct {
         USHORT  ButtonFlags;
         USHORT  ButtonData;
      };
  };
  ULONG  RawButtons;
  LONG  LastX;
  LONG  LastY;
  ULONG  ExtraInformation;
} MOUSE_INPUT_DATA, *PMOUSE_INPUT_DATA;
```

LastX and LastY indicate the increment of mouse movement in horizontal and vertical directions respectively. ButtonFlags tells the state transition of mouse buttons in the following macros.

MOUSE_LEFT_BUTTON_DOWN	The left mouse button changed to down.
MOUSE_LEFT_BUTTON_UP	The left mouse button changed to up.
MOUSE_RIGHT_BUTTON_DOWN	The right mouse button changed to down.
MOUSE_RIGHT_BUTTON_UP	The right mouse button changed to up.
MOUSE_MIDDLE_BUTTON_DOWN	The middle mouse button changed to down.

MOUSE_MIDDLE_BUTTON_UP	The middle mouse button changed to up.
MOUSE_BUTTON_4_DOWN	The fourth mouse button changed to down.
MOUSE_BUTTON_4_UP	The fourth mouse button changed to up.
MOUSE_BUTTON_5_DOWN	The fifth mouse button changed to down.
MOUSE_BUTTON_5_UP	The fifth mouse button changed to up.
MOUSE_WHEEL	Mouse wheel data is present.
MOUSE_HWHEEL	Mouse horizontal wheel data is present.

On the other hand, the data structure for simulating a mouse event is defined as:

```
typedef struct tagMOUSEINPUT {
    LONG dx;
    LONG dy;
    DWORD mouseData;
    DWORD dwFlags;
    DWORD time;
    ULONG_PTR dwExtraInfo;
} MOUSEINPUT, *PMOUSEINPUT;
```

Where dx and dy indicate the cursor movement and dwFlags indicates the status of buttons. The keyboard data also needs to be converted. The data structure read from the keyboard filter driver is:

```
typedef struct _KEYBOARD_INPUT_DATA {
  USHORT  UnitId;
  USHORT  MakeCode;
  USHORT  Flags;
  USHORT  Reserved;
  ULONG  ExtraInformation;
} KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

The data structure for simulating a keyboard event is:

```
typedef struct tagKEYBDINPUT {
    WORD wVk;
    WORD wScan;
    DWORD dwFlags;
    DWORD time;
    ULONG_PTR dwExtraInfo;
} KEYBDINPUT, *PKEYBDINPUT;
```

The wScan variable in KEYBDINPUT needs to be assigned with the value of MakeCode in KEYBOARD_INPUT_DATA.

A software cursor is in the interaction mode by default. It enters the annotation mode by clicking the middle mouse button. When the annotation mode starts, a full screen image is captured and put into a memory device context called memDC. (memDC have a canvas of the same size as the full screen.) The content of memDC is then saved into a file. In the annotation mode, a user can scribble by holding down the left button. Whenever the cursor moves with the left button down, a line will be drawn directly into the screen with the device context obtained with the GetDC(NULL) function and into memDC. To turn of the annotation mode and go back to the interaction mode, the user only needs to click the middle button again. At that time, The content of memDC will be saved again into a different file as the result of the annotation.

6. Copy and Paste between Two Computers

A software cursor corresponds to an actual mouse and is uniquely identifiable. A user can copy an item from an application running on a computer and store it with the software cursor. Then the user can move the software cursor to another computer and paste the stored item to the second application on the second computer. Currently we can copy both text and files between two computers.

6.1 Keep Clipboard with Software Cursor

In a today's operation system, the clipboard is associated with a computer so that multiple applications running on the same computer can exchange the information through the shared clipboard. In our implementation, a user can operate a software cursor and a clipboard can be associated with a software cursor. The clipboard associated with a computer is referred to as a *machine clipboard*, and the one with a cursor is called a *cursor clipboard* hereafter. Since we made no changes to the operating system or the applications running on the operating system, our cursor clipboard implementation is still based on the machine clipboard. Whenever the machine clipboard is changed, its new content may be copied to the cursor clipboard of the software cursor active on the machine. To paste the content of a cursor clipboard to an application, the cursor clipboard content has to be copied to the machines clipboard first. When a clipboard is kept with a software cursor, different software cursors can have different clipboards. Figure 6.1 illustrates three software cursors with three different clipboards.

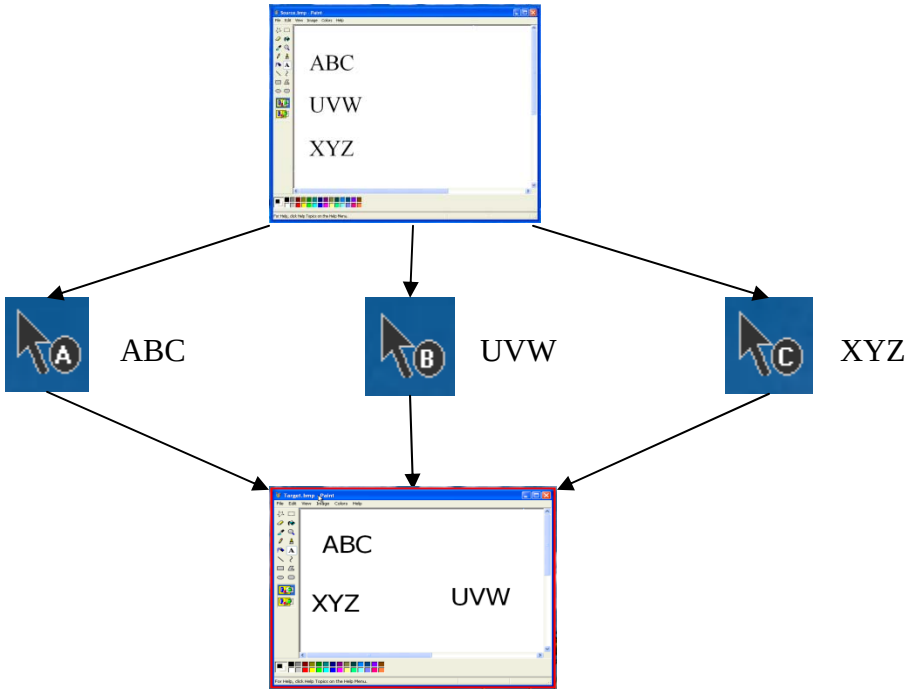

Figure 6.1 Three software cursors with three different clipboards.

To monitor whether there is any change to the machine clipboard, we added the software cursor as a clipboard viewer using the *SetClipboardViewer* function. Before the software cursor is destroyed, it needs to be removed from the chain by calling the *ChangeClipboardChain* function. When the software cursor is in the chain, it must process the clipboard messages *WM_CHANGECBCHAIN* and *WM_DRAWCLIPBOARD*, and call the *SendMessage* function to pass these two messages to the next window in the clipboard viewer chain.

Whenever there is a change to the machine clipboard, the *WM_DRAWCLIPBOARD* message will be sent to the clipboard viewer. The clipboard viewer can retrieve the content of the machine clipboard and try to copy it to the cursor clipboard if it determines that the machine clipboard was not changed by itself using the *GetClipboardOwner* function. Whenever a user clicks a software cursor on a computer, the software cursor will be actively associated with the corresponding machine if the association does not exist before. At the same time, the content of the cursor clipboard will be sent to the corresponding machine clipboard. After that, the paste operation will use the new machine clipboard content which is the same as the cursor clipboard of the active software cursor.

6.2 Copy Files between Two Computers

With the software cursor, we can copy files between two machines. A user can accomplish file copying in the following steps.

1). Open a Windows Explorer on the source machine.
2). Select files and/or directories to be copied.
3). Copy (using *Ctrl+C* for example) the selected items to the software cursor.
4). Move the software cursor to the target machine.
5). Open a Windows Explorer on the target machine.
6). Go to the target directory.
7). Paste (using *Ctrl+V* for example) the files and/or directories recorded on the software cursor to the target directory.

To copy and paste files between two computers, we define the data format for the cursor clipboards as shown in Figure 6.2. The *count* field indicates the number of formats for the clipboard. For each format, it consists of the format name, the length of the data, and the actual data. Currently, we implemented two formats, *CF_TEXT* and *CF_HDROP*. The *CF_TEXT* format is for text copy and paste, and its data is a NULL-terminated string. The *CF_HDROP* format is for file copy and paste, and its data is a list of files and directories. Each file or directory is NULL-terminated and the list is also NULL-terminated. Therefore, the *CF_HDROP* data is composed of multiple NULL-delimited strings that are ended with a double NULL terminator. The *IP_Addr* field indicates the clipboard source. Later, the target machine can use the *IP_Addr* and *Port* fields to retrieve the files using the protocol described in Section 6.3.

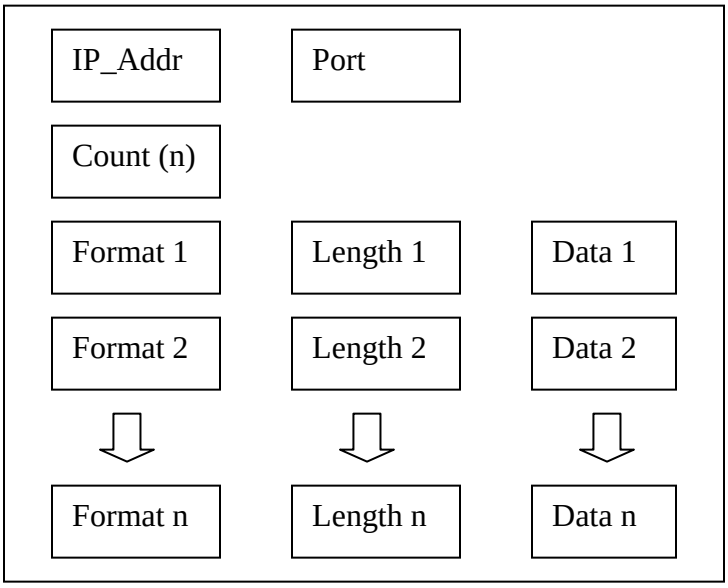

Figure 6.2 The data structure for cursor clipboards.

Whenever there is a change to the machine clipboard, the *WM_DRAWCLIPBOARD* message will be sent to all the clipboard viewers. Our clipboard viewer will enumerate all the formats using *EnumClipboardFormats*. We are only interested in two formats, *CF_TEXT* and *CF_HDROP*. If the format is of *CF_HDROP*, we use the *GetClipboardData* function to get the *HDROP* handle, and then use the *DragQueryFile* function to get all the path names from the *HDROP* handle. The path names along with the machine IP address and the port number of the file transfer service will be packed as shown in Figure 6.2 and stored with the software cursor. If a user selects a list of files and directories and performs a copy operation in Windows Explorer, the path names for these files and directories along with the machine IP address and the service port number will be saved into the cursor clipboard under the *CF_HDROP* format.

Whenever the software cursor is associated with a new computer, the software cursor needs to set the cursor clipboard into the machine clipboard. However, each association does not always lead to a paste operation. If there is no paste operation, there is no need to transfer files. To be efficient, we adopted delayed rendering. Specifically, we use *SetClipboardData* to set the *CF_HDROP* data to NULL. Whenever there is a paste operation, the *WM_RENDERFORMAT* message with the *CF_HDROP* format will be sent to the owner. The owner then can make a connection to the server using the IP address and the service port number. Once the connection is established, the owner (i.e. the client) can send the pathnames one by one to request the files using the simple file transfer protocol described in Section 6.3. The received files will be saved in a temporary directory. At the same time, all the pathnames are mapped to reflect their new locations under the temporary directory. The mapped path names prefixed by a *DROPFILES* structure will then be set into the *CF_HDROP* format of the machine clipboard using *SetClipboardData*. At this time, the corresponding files will be transferred from the temporary directory into the target directory. Therefore, if a user performs a paste operation in Windows Explorer, the files recorded in the corresponding software cursor will be transferred from the source to the target machine and directory. This is how the files are pasted to the target machine and directory using the software cursor.

Whenever there is an owner change to the machine clipboard, made by the new owner using the *EmptyClipboard* function, the *WM_DESTROYCLIPBOARD* message is sent to the previous owner. At that time, the previous owner can delete the files under the temporary directory.

6.3 Simple File Transfer Protocol

To facility file transferring between two computers, we implemented a simple file transfer protocol using TCP/IP. The file transfer server needs to run on any physical computer within a virtual computer because any computer with the virtual computer can be the source of file transferring. The server listens on a default port of 10004. When a

client makes a connection to the server, the server will spawn a thread to handle the file transfer requests from the client. The file transfer request is of the (*PATH_NAME, length, pathname*) form. The *pathname* can be a file name or a directory name on the server. The client does not need to know the type of *pathname*. If the *pathname* is determined to be a file at the server, the server will retrieve the file and send it to the client. Specifically, the reply starts with the (*FILE_NAME, length, filename*) packet followed by a number of (*DATA_FILE, length, data*) packets depending on the file size. Please note that the last *DATA_FILE* packet always has *length*=0 to indicate the end of the file. To be consistent with the directory transfer case, the server sends the *END_LIST* packet that carries no parameter to indicate the end of reply. When the client receives the *FILE_NAME* packet, it splits the *filename* into the directory and file parts, and then creates the directory under a temporary directory using *SHCreateDirectoryEx* and creates the file using *CreateFile*. Please note that *CreateDirectory* may not work because some immediate directories have to be created too. The client will write the received *DATA_FIL*E packets to the file until the (*DATA_FILE, 0*) packet arrives. At that time, the client closes the file.

If the *pathname* in (*PATH_NAME, length, pathname*) is a directory, the server will retrieve and send the whole sub-tree rooted at *pathname*. The server uses the *FindFirstFile, FindNextFile* and *FindClose* functions to enumerate all the files and subdirectories under a requested directory. Please note that the files and directories enumerated are relative names and include two special directories "**.**" and "**..**". For each file, the server will send the file using the protocol described in the previous paragraph, i.e., a *FILE_NAME* packet followed by a bunch of *DATA_FILE* packets. For each directory other than "**.**" or "**..**", the server will send (*DIR_NAME, length, dirname*) packet, and recursively enumerate all the files and subdirectories under the directory of *dirname*. On receiving the *DIR_NAME* packet, the client will create a corresponding directory using *SHCreateDirectoryEx*. The *END_LIST* packet is sent by the server to indicate the end of reply.